



# 您身边的金牌教练

沈文选	教授	金牌教练	湖南师范大学
唐立华	特级教师	金牌教练	华东师范大学附中
冯志刚	特级教师	金牌教练	上海中学
冯跃峰	特级教师	金牌教练	深圳高级中学
王树国	高级教师	金牌教练	湖南师范大学附中
黄生训	教授	金牌教练	湖南师范大学
武建谋	特级教师	金牌教练	长沙市一中
刘旭华	高级教师	金牌教练	湖南师范大学附中
黄洪才	高级教师	金牌教练	长沙市一中
彭大斌	特级教师	金牌教练	长沙市一中
邓立新	特级教师	金牌教练	长沙市一中
陈云莎	特级教师	金牌教练	湖南师范大学附中
肖鹏飞	特级教师	金牌教练	湖南师范大学附中
高建军	特级教师	金牌教练	长沙市一中
黄国强	特级教师	金牌教练	湖南师范大学附中
汪训贤	特级教师	金牌教练	湖南师范大学附中
吴耀斌	副教授	金牌教练	中南大学
向期中	高级教师	金牌教练	长郡中学
曹利国	高级教师	金牌教练	长沙市一中



#### ◆ 向期中

长郡中学特级教师，湖南省计算机学会理事，国际金牌教练，国家教育部计算机课程咨询委员会委员。对中小学计算机教育事业有一种执着的追求，参加工作20年来，一直以“当一流教师，办一流教育，出一流人才”为自己的工作目标，对中小学计算机教学和青少年信息学奥林匹克竞赛的辅导倾注了全部热情和心血。在信息学奥林匹克竞赛培训中把“先做人，后成才”的育人理念贯穿到整个奥赛培训的始终，学生在愉快的学习中取得了一个个辉煌的成绩：在近几年的信息学奥林匹克竞赛中，辅导的学生有100多人获湖南省一等奖，11人次进入国家集训队，3人进入国家代表队，3人获国际金牌。撰写了《信息学（计算机）国际奥林匹克Turbo Pascal 6.0》等十多部信息学专著。多次荣获园丁奖和全国优秀辅导员称号，还先后获得全国中小学计算机教育先进工作者、湖南省优秀教师和全国信息学奥林匹克竞赛高级指导教师等荣誉称号。

普及信息技术  
提高青少年  
科学素质

祝《奥赛经典丛书》出版

陈火旺

## 前言

国际信息学奥林匹克竞赛 (IOI) 是计算机知识在世界范围青少年中普及的产物。它始于 1989 年, 是继数学、物理和化学之后的又一门国际 (中学生) 学科奥林匹克竞赛。在国际学科奥林匹克竞赛中, 我国只有信息学是在 1989 年首次 IOI 中就具有参赛资格的, 而且首届竞赛的试题原型是由我国提供的。

20 世纪 80 年代, 邓小平同志在视察青少年校外计算机活动时指出: “计算机的普及要从娃娃抓起。”从此, 全国性的青少年计算机竞赛活动每年都吸引着数以万计的青少年投身到这一活动中, 也成为我国校外计算机活动中最有代表性的形式。竞赛是青少年喜闻乐见的课外活动形式, 但竞赛不是目的, 只是推广、普及的一种手段, 而普及计算机知识则是我国的国策, 也是世界发展的趋势。培养高素质的信息技术人才, 才是竞赛的最终目的。

为了进一步推广、普及计算机技术, 提高竞赛水平, 在原来编写的一套《信息学奥林匹克教程》(基础篇·提高篇·语言篇) 的基础了, 我们又编写了这本《数据结构篇》。

《数据结构篇》主要帮助学生全面地掌握数据结构知识与应用技巧, 相对于其他数据结构书不同之处就在于增加了一些针对性的例题和习题, 着眼点是提高数据结构的应用方法与技巧, 是一本具有实战意义的教材。

从逻辑角度看, 数据可归结为三种基本结构: 线性结构、树结构和图结构; 从存储角度看, 数据可归结为四种基本结构: 顺序结构、链接结构、索引结构和散列结构。每一种逻辑结构可根据不同需要采用不同的存储结构, 或者不同的存储结构的组合。数据的逻辑结构和存储结构确定后, 再结合指定运算的算法, 就容易利用一种程序设计语言编写出程序。通过数据结构的学习, 能够大大提高程序设计能力和水平。

《数据结构篇》是为广大信息学爱好者学习数据结构而精心编著的一本教材。本书内容比较全面, 着重于实用与实战, 在算法分析上简明扼要, 细致清晰, 便于自学。全书共分十章: 第一章为概论, 它为学习以后的各章做准备; 第二章至第五章为线性结构; 第六章和第七章分别为树结构和图结构, 分别讨论了每一种逻辑结构所对应的存储结构和相应的算法; 第八章和第九章分别为查找与排序, 它包含了数据处理中主要使用的几种查找和内排序方法; 最后一章为读者提供了

检测知识的模拟试题及解答。本书讲授时数为 80 学时左右。

本书的算法主要是以 PASCAL 或类 PASCAL 为基础编写实现的。

本书同时也适合作大学生及研究生的参考资料。

参加编写本书的有李明威、谢秋锋、石东妮、刘涛、曾文武、胡伟栋、王俊、任恺等，特别感谢周戈林、郭华阳、杨浩、周玉姣等同学对本书提供的帮助。

本书所引用的试题凝聚了国内外多年来积极参与青少年信息学奥林匹克竞赛命题工作的专家、教授的心血和劳动，许多参赛选手的解题思想、方法和技巧给予了我们极大的启发和借鉴。本书得到了湖南师范大学出版社的大力支持和帮助，在此表示衷心的感谢！

由于水平和时间有限，不妥之处在所难免，敬请读者批评指正。

编者

2006 年 7 月

## 目 录

1 概论 .....	(1)
1.1 基本术语 .....	(1)
1.2 算法描述 .....	(6)
1.3 算法评价 .....	(9)
1.4 Pascal 语言中的数据类型 .....	(14)
1.5 小结 .....	(17)
习题一 .....	(18)
2 线性表 .....	(21)
2.1 线性表的定义和顺序存储 .....	(21)
2.2 线性表的运算 .....	(22)
2.3 线性链表及链接存储 .....	(26)
2.4 线性表的应用举例 .....	(32)
2.5 小结 .....	(35)
习题二 .....	(36)
3 栈和队列 .....	(41)
3.1 栈 .....	(41)
3.2 栈的应用举例 .....	(45)
3.3 队列 .....	(53)
3.4 队列的应用举例 .....	(56)
3.5 链接的栈和队列 .....	(58)
3.6 小结 .....	(61)
习题三 .....	(61)
4 串 .....	(65)
4.1 串的基本概念 .....	(65)
4.2 串的定义 .....	(65)
4.3 串的实现及基本运算 .....	(66)
4.4 串的应用 .....	(69)
4.5 小结 .....	(81)
习题四 .....	(82)
5 数组、特殊矩阵和广义表 .....	(87)
5.1 多维数组 .....	(87)
5.2 稀疏矩阵 .....	(88)
5.3 特殊矩阵的压缩存储 .....	(91)

5. 4	广义表	(92)
5. 5	小结	(94)
	习题五	(94)
6	树	(96)
6. 1	树的概念	(96)
6. 2	二叉树	(100)
6. 3	二叉树的运算	(106)
6. 4	二叉搜索树	(108)
6. 5	哈夫曼树	(113)
6. 6	树的存储结构和运算	(116)
6. 7	树、森林和二叉树的转换	(119)
6. 8	最近公共祖先	(119)
6. 9	树状数组	(122)
6. 10	并查集	(125)
6. 11	树的应用举例	(129)
6. 12	小结	(156)
	习题六	(156)
7	图	(162)
7. 1	图的概念	(162)
7. 2	图的基本术语	(163)
7. 3	图的存储结构	(165)
7. 4	图的遍历	(168)
7. 5	图的生成树与最小生成树	(171)
7. 6	最短路径	(178)
7. 7	拓扑排序	(188)
7. 8	关键路径	(193)
7. 9	图的应用举例	(199)
7. 10	小结	(211)
	习题七	(211)
8	查找	(216)
8. 1	查找的基本概念	(216)
8. 2	顺序表查找	(217)
8. 3	索引查找	(221)
8. 4	散列查找	(223)
8. 5	树表查找	(229)
8. 6	查找的应用举例	(239)
8. 7	小结	(254)
	习题八	(254)
9	排序	(258)



9.1	排序的基本概念 .....	(258)
9.2	简单排序算法 .....	(259)
9.3	快速排序 .....	(263)
9.4	堆排序 .....	(266)
9.5	归并排序 .....	(270)
9.6	各种排序方法比较 .....	(272)
9.7	线性时间排序 .....	(273)
9.8	排序的应用举例 .....	(278)
9.9	小结 .....	(282)
	习题九 .....	(282)
10	模拟试题 .....	(285)
10.1	数据结构综合测试一 .....	(285)
10.2	数据结构综合测试二 .....	(291)
10.3	数据结构综合测试三 .....	(297)
10.4	数据结构综合测试四 .....	(299)
10.5	数据结构综合测试一参考答案 .....	(302)
10.6	数据结构综合测试二参考答案 .....	(303)
10.7	数据结构综合测试三参考答案 .....	(305)
10.8	数据结构综合测试四参考答案 .....	(313)
	习题参考答案 .....	(319)



# 1 概 论

自1946年美国第一台电子计算机问世以来,计算机科学和软硬件得到了飞速的发展,与此同时,计算机应用领域也从最初的科学计算逐步发展到人类活动的各个领域。现在,计算机处理的对象不仅是简单的数值或字符,而且是带有不同结构的各种数据:图像、声音等。因此,要设计出一个较好的程序,除了掌握所用的计算机语言外,还要研究各种数据结构的特性和数据之间存在的关系,这就是“数据结构”这门学科形成和发展的背景。

要搞好信息学竞赛,最基本的就是要掌握好程序设计,而程序设计是一门综合学科,与程序设计最密切的课程有数据结构、算法分析与设计和程序设计方法学等。著名的计算机科学家沃斯(N. Wirth)甚至提出了“算法+数据结构=程序”的著名论点,简明地概括了程序的组成。

数据是程序加工的原材料,它可能是数字、字符或由它们组成的字符串;它也可能是采样后的物理量,例如电压、电流等电信号通过模-数转换器(A/D)输出变成计算机可以接受的数字信息;或是从磁带、磁盘和光盘上读出的--串二进制数表示的数字、字符或图形的信息;或是调制解调器(Modem)上将电话声音信号转换成计算机可以接受的格式;或是通过键盘、磁盘文件输入到计算机的信息……简言之,数据是描述客观事物的数字、字符以及所有能输入到计算机、能被计算机进行处理的信息集合。也就是说,数据是符号的集合,是计算机要处理的信息集合。从本质上来说,数据是客观事物表示的一种抽象结果,而数据结构课程就是研究如何把客观世界要处理的信息逐层抽象成计算机可以接受的某种形式。

算法是解题的方法和步骤的精确描述,它是有穷处理的序列。

数据结构和算法有着密切的联系,数据结构是建立在算法的基础上,而选择什么样的数据结构对于程序设计来说,是至关重要的决策,它直接影响到程序的效率。选择一个合适的数据结构便很容易形成一个简洁有效的算法;否则,如果数据结构选择不好,除了影响程序开发速度之外,更重要的是影响设计出来的程序的运行效率。

## 1.1 基本术语

这一节将对全书中常用的名字和术语赋予确定的含义,便于对书本的阅读理解。

数据(Data)是人们利用文字符号、数字符号以及其他规定的符号对现实世界的事物及其活动所做的描述。因此,大到一本书,一篇文章,一张图表等数据,小到--个句子,一个单词,一个算式,一个数字和一个字符等,总之,数据是信息的载体。人们把能够被计算机识别、输入、存储、处理和输出的一切信息都叫数据。

数据元素(Data Element)是一个数据整体中相对独立的单位。如对于一个文件,每个记录就是它的数据元素;对于一个字符串,每个字符就是它的数据元素;对于一个数组,每一个分量就是它的数据元素。数据和数据元素是相对而言的,如对于一个记录,它相对于所在的文件被认为

是数据元素，而相对于它所含的数据项（域）又被认为是数据。因此，本书中对数据和数据元素这两个术语的使用并不加以严格的区分。

数据记录（Data Record）简称记录，它是数据处理领域组织数据的基本单位。它又由更小的单位——数据项（Item，或称为域）所组成，一个记录一般包括一个或若干个固定的数据项（当然每一个数据项还可以是记录形式）。如下表 1-1 就是一个班的学生期中考试的成绩表，每个记录表示一个学生的考试基本情况。

表 1-1 0301 班学生成绩表

学号	姓名	语文	数学	外语	物理	化学	总分
0301001	陈天宇	86	92	88	81	89	436
0301002	刘思佳	85	92	84	86	96	443
0301003	夏俊伟	82	93	86	88	93	442
0301004	黄艺海	80	95	81	89	90	435
0301005	何 轩	88	90	90	90	87	445
0301006	邓亦龙	84	98	87	93	86	448
0301007	袁浩翰	83	94	90	93	88	448
0301008	肖湘宁	88	88	96	82	88	442
.....							

数据处理（Data Processing）是指对数据进行查找、插入、删除、合并、排序、统计、简单计算、输入、输出等的操作过程。在早期，计算机主要用于科学和工程计算，进入 20 世纪 80 年代以后，计算机主要用于数据处理。据有关统计资料表明，现代计算机用于数据处理的时间比例平均高达 80% 以上。随着时间的推移和计算机的进一步普及，计算机用于数据处理的时间比例必将进一步增大，像计算机情报检索系统、经济管理信息系统、图书管理系统、银行核算系统、财务管理系统、招生及成绩管理系统等都是计算机在数据处理领域的具体应用。数据结构是数据处理软件的基础，因此数据结构课程是计算机所有专业的最重要的主干课程之一。

数据结构（Data Structure），简单地说是指数据以及数据之间的联系。上面提到数据的描述对象是现实世界的事物及其活动，而任何事物及其活动都不是孤立存在的，都是在一定意义上相互联系、相互影响的，所以数据之间必然存在着联系。由于这种联系是内在的，或根据需要人为定义的，所以被认为是“逻辑”上的联系，并把数据结构作为数据的逻辑结构。数据结构在计算机存储器上的存储表示称作为数据的物理结构或存储结构。由于存储表示方法有顺序、链接、索引、散列等多种，所以一种数据结构可表示成一种或多种物理结构。确切地说，数据结构就是研究数据和数据之间的逻辑结构和物理结构，而重点研究它们的逻辑结构。

为了更准确地描述数据结构，我们采用二元组表示：

$$B = (K, R)$$

B 是一种数据结构，它由数据元素集合 K 和 K 上二元关系的集合 R 所组成。

$$K = \{k_i \mid 1 \leq i \leq n, n \geq 0\}$$

$$R = \{r_j \mid 1 \leq j \leq m, m \geq 0\}$$

$k_i$  表示第  $i$  个数据元素,  $n$  为  $B$  中数据元素的个数, 特别地, 若  $n=0$ , 则  $K$  是一个空集, 因而  $B$  也无结构可言, 或者说它具有任何结构;  $r_j$  表示第  $j$  个二元关系 (以后简称关系),  $m$  为  $K$  上关系的个数。

本书讨论的数据结构, 一般只讨论  $m=1$  的情况, 即  $R$  中只包含一个关系  $R = \{r\}$  的情况, 对于包含多个关系的数据结构, 可分别对每一个关系进行讨论。

$K$  上的一个关系  $r$  是序偶集合。对于  $r$  中的任一序偶  $\langle x, y \rangle$  ( $x, y \in K$ ), 把  $x$  叫做序偶的第一个元素,  $y$  叫做序偶的第二个元素, 又称序偶的第一个元素为第二个元素的直接前驱, 简称前驱, 称第二个元素为第一个元素的直接后继, 简称后继。如在序偶  $\langle x, y \rangle$  中,  $x$  为  $y$  的前驱, 而  $y$  为  $x$  的后继。

为了更好地理解, 数据结构我们习惯用图形形象地表示出来, 图形中的每个结点 (或叫顶点, 或叫节点) 对应着一个数据元素, 两结点之间带箭头的连线 (称作有向边或弧) 对应着关系中的一个序偶, 其中序偶的第一个元素为有向边的起始结点, 第二个元素为有向边的终止结点。

下面通过一个具体实例, 根据表 1-2 构造一些典型的数据结构。

表 1-2 信息学竞赛培训名单

编 号	姓 名	性 别	出生年月日	职 务	单 位
01	向期中	男	1965 年 10 月	教练	
02	金 恺	男	1986 年 9 月 25 日	组长	高三
03	栗 师	男	1986 年 9 月 11 日	学员	高三
04	柳明海	男	1986 年 8 月 11 日	学员	高三
05	任 恺	男	1987 年 4 月 17 日	组长	高二
06	王 俊	男	1986 年 8 月 1 日	学员	高二
07	胡伟栋	男	1986 年 8 月 5 日	学员	高二
08	易 伟	男	1986 年 12 月 19 日	学员	高二
09	康亮环	男	1986 年 12 月 14 日	学员	高二
10	周戈林	男	1987 年 12 月 30 日	组长	高一
11	肖湘宁	女	1988 年 10 月 16 日	学员	高一
12	谭 欣	女	1988 年 12 月 28 日	学员	高一
13	邓亦龙	男	1988 年 9 月 30 日	学员	高一

表中共有 13 条记录, 每条记录都由六个数据项所组成, 由于每条记录的编号各不相同, 所以可把每条记录的编号作为该记录的关键字。在下面的例子中, 我们用记录的关键字代表整个记录。

例题 1-1 一种数据结构  $\text{linearity} = (K, R)$ , 其中

$$K = \{01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12, 13\}$$

$R = \{r\}$

$r = \{ \langle 01, 06 \rangle, \langle 06, 07 \rangle, \langle 07, 04 \rangle, \langle 04, 03 \rangle, \langle 03, 02 \rangle, \langle 02, 09 \rangle, \langle 09, 08 \rangle, \langle 08, 05 \rangle, \langle 05, 10 \rangle, \langle 10, 13 \rangle, \langle 13, 11 \rangle, \langle 11, 12 \rangle \}$

对应的图形如图 1-1 所示:

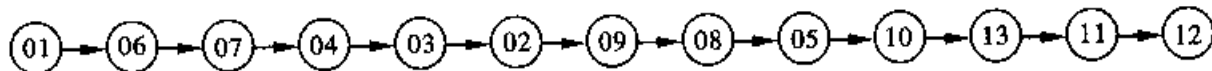


图 1-1 线性结构示意图

结合表 1-2, 不难发现,  $r$  是按年龄从大到小的排列关系。

在 linearity 中, 每个数据元素有且只有一个直接前驱元素 (除结构中第一个元素 01 外), 有且只有一个直接后继元素 (除结构中最后一个元素 12 外)。这种数据结构的特点是数据元素之间的 1:1 关系, 即线性关系, 把具有这种特点的数据结构叫线性结构。

**例题 1-2** 一种数据结构  $tree = (K, R)$ , 其中

$K = \{01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12, 13\}$

$R = \{r\}$

$r = \{ \langle 01, 02 \rangle, \langle 01, 05 \rangle, \langle 01, 10 \rangle, \langle 02, 03 \rangle, \langle 02, 04 \rangle, \langle 05, 06 \rangle, \langle 05, 07 \rangle, \langle 05, 08 \rangle, \langle 05, 09 \rangle, \langle 10, 11 \rangle, \langle 10, 12 \rangle, \langle 10, 13 \rangle \}$

对应的图形如图 1-2 所示:

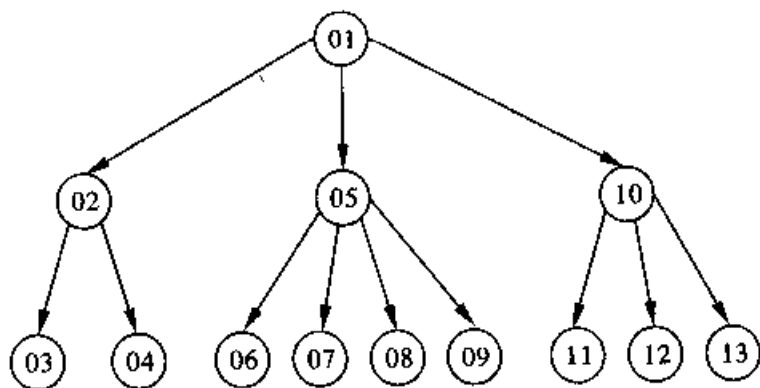


图 1-2 树结构示意图

结合表 1-2, 不难发现,  $r$  是教练、学员之间管理与被管理之间的关系。

图 1-2 像倒着的一棵树, 在这棵树中, 最上面的一个没有前驱只有后继的结点叫作根结点, 最下面一层的只有前驱没有后继的结点叫作树叶结点, 除树根结点和树叶结点之外的结点叫做树枝结点。在一棵树中, 每个结点有且只有一个前驱结点 (除树根结点外), 但可以有任意多个后继结点 (树叶结点可看作具有 0 个后继结点)。这种数据结构的特点是数据元素之间的 1: N 关系 ( $N \geq 0$ ), 把具有这种特点的数据结构叫作树型结构或树结构。

**例题 1-3** 一种数据结构  $graph = (K, R)$ , 其中

$K = \{01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12, 13\}$

$R = \{r\}$

$r = \{ \langle 01, 02 \rangle, \langle 01, 03 \rangle, \langle 01, 04 \rangle, \langle 01, 05 \rangle, \langle 01, 13 \rangle, \langle 01, 10 \rangle, \langle 05, 06 \rangle, \langle 05, 07 \rangle, \langle 05, 08 \rangle, \langle 05, 09 \rangle, \langle 10, 11 \rangle, \langle 10, 12 \rangle, \langle 10, 13 \rangle \}$

$\langle 06 \rangle, \langle 05, 07 \rangle, \langle 06, 08 \rangle, \langle 08, 09 \rangle, \langle 05, 13 \rangle, \langle 10, 11 \rangle, \langle 11, 10 \rangle, \langle 11, 12 \rangle$

对应的图形如图 1-3 所示:

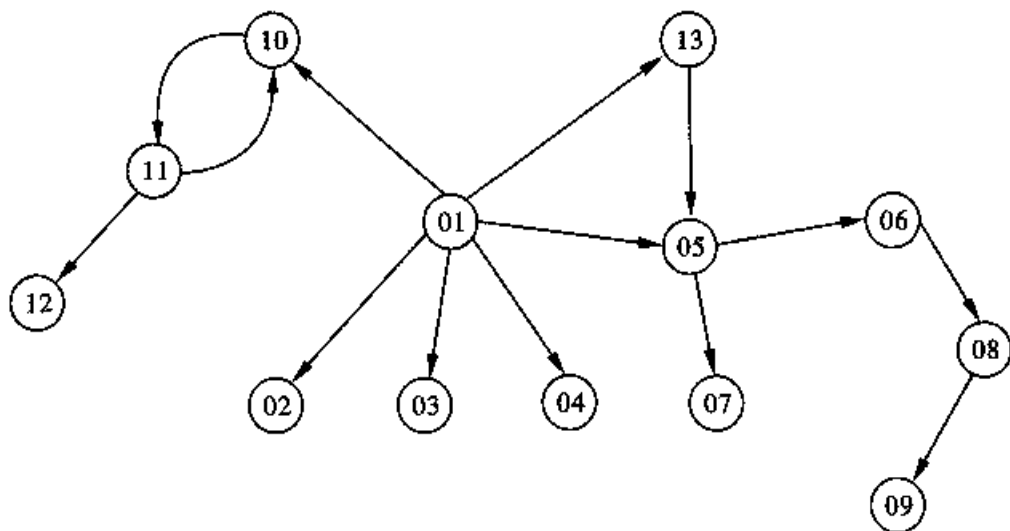


图 1-3 图型结构示意图

从图 1-3 可以看出,  $r$  是  $K$  上一个关系, 不妨叫教练与学员以及学员之间的熟知关系, 得到的是一个有向图。若再定义一个关系  $r$ , 为每个序偶元素的友好关系, 那么得到图 1-4, 由于友好关系是相互的, 显然是一个无向图。

$r = \{ \langle 01, 02 \rangle, \langle 01, 05 \rangle, \langle 01, 13 \rangle, \langle 02, 03 \rangle, \langle 02, 04 \rangle, \langle 02, 05 \rangle, \langle 03, 04 \rangle, \langle 03, 07 \rangle, \langle 03, 13 \rangle, \langle 04, 05 \rangle, \langle 04, 06 \rangle, \langle 04, 07 \rangle, \langle 05, 06 \rangle, \langle 05, 11 \rangle, \langle 06, 07 \rangle, \langle 06, 08 \rangle, \langle 06, 10 \rangle, \langle 06, 11 \rangle, \langle 08, 09 \rangle, \langle 08, 12 \rangle, \langle 09, 10 \rangle, \langle 10, 12 \rangle \}$

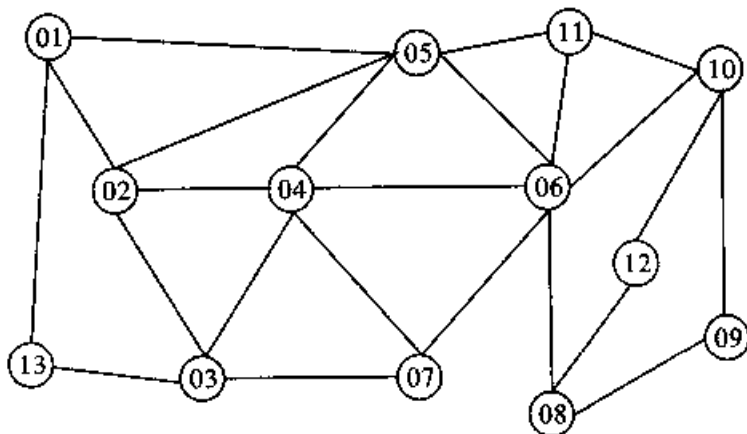


图 1-4 图型结构示意图

从图 1-3 和 1-4 可以看出, 结点之间的联系是  $M:N$  的 ( $M \geq 0, N \geq 0$ ), 也就是说每个结点可以有任意多个前驱结点和任意多个后继结点。我们把具有这种特点的数据结构叫做图型结构。

从图型结构、树型结构和线性结构的定义可知, 树型结构是图型结构的特殊情况 (即  $M=1$  的情况), 线性结构是树型结构的特殊情况 (即  $N=1$  的情况)。为了区别于线性结构, 人们把树

型结构和图型结构统称为非线性结构。

当然,在所有结构中,当结点的联系取最小值,即  $M=0$ ,  $N=0$  时,每个结点都是孤立的,这时称为离型结构,由于结点间没有建立联系,研究价值不大,本书未做讨论。

特别的一个结构中可以有二种或以上的关系时,就变得很复杂,这时我们解决的办法是将他们分离只含有一种关系进行处理,相当于多个一种结构的并列,因此也可以归结于我们上面讨论的几种结构。

算法 (algorithm), 简明地说就是解决特定问题的方法和步骤。特定的问题可以分为数值的和非数值的两类。解决数值问题的算法叫做数值算法, 科学和工程计算方面的算法属于数值算法, 如求解数值积分, 求解线性方程组, 求解代数方程等。解决非数值问题的算法叫做非数值算法, 数据处理方面的算法都属于非数值算法, 如各种排序算法、查找算法、删除算法、遍历算法等。数值算法和非数值算法并没有严格的区别, 一般来说, 在数值算法中主要进行算术运算, 而在非数值算法中, 则主要进行比较和逻辑运算。另一方面, 特定的问题可能是递归的, 也可能是非递归的, 因而解决它们的算法就有递归算法和非递归算法之分, 当然, 从理论上讲, 任何递归算法都可以通过循环、堆栈等技术转化为非递归算法。

在信息学领域, 一个算法实质上是针对所处理问题的需要, 在数据的逻辑结构和存储结构的基础上施加的一种运算。由于数据的逻辑结构和存储结构不是唯一的, 在很大程度上可以由用户自行选择和设计, 所以处理同一个问题的算法也不是唯一的。另外, 即使对于具有相同的逻辑结构和存储结构而言, 其算法设计的思想和技巧不同, 编写的算法也不再相同。我们学习数据结构这门课的目的, 就是要会根据数据处理问题的需要, 为待处理的数据选择合适的逻辑结构和存储结构, 进而设计出优秀的算法, 从而达到解决问题的目的。

## 1.2 算法描述

鉴于本书的侧重面, 阅读本书是在学习了 Turbo Pascal 语言的前提下, 因此对于书中算法的描述都是采用 Turbo Pascal 的语言语法规则, 同时增加一些功能较强的语句, 有些过程与函数也采用类 Pascal 语言书写, 这样使描述出的算法清晰、直观, 便于阅读和分析。读者不难用 Turbo Pascal 编程实现。

使用的语句如下:

1. 赋值语句

变量名: = 表达式;

2. 读语句

Read (变量名表) 或 Readln < 变量名表 >;

3. 写语句

Write (输出项) 或 Writeln < 输出项 >;

4. 转向语句

Goto 语句标号;

5. 调用过程语句

过程名 (参数表);

## 6. 退出循环语句

Break;

用于各种循环中,该语句被执行时,将立即退出当前循环,相当于 goto 到该循环语句后的第一条语句上,它是循环语句的一个非正常出口。

## 7. 返回语句

Exit; Return 或 Return 参数;

用于过程或函数体中,该语句被执行时,将立即退出当前过程或函数,返回到调用前所保存的位置继续向下执行,它是执行过程或函数的一个非正常出口。

## 8. 非正常结束程序语句

Halt;

在程序的任何一个地方使用该语句,则立即结束程序。

## 9. 出错处理语句

Error (字符串);

在算法中为了避免非法操作,需要进行出错处理时统一使用该语句,它表示此算法执行到此中止。在实际算法中,它可能是转去执行一个错误处理程序,也可能是简单地打印出错误信息并停止运行等。该语句括号内的字符串用以说明出错的原因,如使用字符串“Overflow”表示溢出,“Out of range”表示超出范围或越界。在 Turbo Pascal 中一般可以定义一个这样的过程来实现。

## 10. 复合语句

Begin 语句 1; 语句 2; ...; 语句 n end;

## 11. 条件语句

If 条件 then 语句 1 [else 语句 2];

条件语句当然还可以嵌套。

## 12. 情况语句

Case 选择表达式 of

常量 1: 语句 1;

常量 1: 语句 2;

.....

常量 n: 语句 n;

[else 语句 n + 1]

End;

## 13. For 循环语句

格式 1:

For 变量名: = 初值 to 终值 do 语句;

格式 2:

For 变量名: = 初值 downto 终值 do 语句;

如果格式 1 中初值大于终值,或格式 2 中初值小于终值则都不会执行循环体。

## 14. While 循环语句

While 条件 do 语句;

## 15. Repeat/Until 循环语句

Repeat 一组语句 Until 条件;

描述算法的书写规则如下:

1. 一般算法采用过程或函数形式书写;为便于读者领会,有时也根据需要写出一个完整的程序。

2. 参数表中的参数如没有类型说明,则规定大写字母或大写字母开头的标识符表示变量形参,小写字母或以小写字母开头的标识符表示值形参。

3. 过程和函数体中的定义和说明部分为了简便有时可能省略;语句部分通常按内容层次分别对语句进行编号。第一层编号用 (1)、(2)、(3)……表示,第二层编号用 (I)、(II)、(III)……表示,第三层编号用 (a)、(b)、(c)……表示,并规定除第一层编号的外面不省略语句括号 begin 和 end 外,其余层次均省略。

例如:对于一维整型数组 A (1:n),其中 1 和 n 分别表示该数组下标的下界和上界,若要求从下标 1 至 n ( $n \geq 1$ ) 的元素中查找出最大值和最小值,并把它们分别赋给 Max 和 Min,则算法描述如下:

Procedure find (A, n, Max, Min);

Begin

(1) Max: = A [1]; Min: = A [1] } 赋初值

(2) For i: = 2 to n do { 查找过程

(I) if A [i] > Max then Max: = A [i];

(II) if A [i] < Min then Min: = A [i];

end;

4. 在条件或循环语句中,出现多条简单语句时,为了简单起见,也不采用编号,而采用方括号代替 begin 和 end 语句括号。如:

If R [i] = T [j] then [i: = i+1; j: = j+1]

Else [i: = i-j+1; j: = 1];

5. 当需要从若干个表达式中取其值最大者或最小者时,可简记为:

Max (表达式 1, 表达式 2……);

Min (表达式 1, 表达式 2……);

实际上这两个函数可以通过条件语句的嵌套或其他方法实现。

6. 当两个变量 X 和 Y 需要相互交换值时,可简记为:

$X \leftrightarrow Y$ ;

实际上它对应下面三条赋值语句:

T: = X; X: = Y; Y: = T; {T 为辅助变量}

给变量 X 赋值有时记为:  $X \leftarrow Y$ ;

此外,在用 Turbo Pascal 描述算法时,还需要用到 Turbo Pascal 中的所有数据类型、标准过程和函数等。



## 1.3 算法评价

对于解决同一个问题,往往能够编写出许多不同的算法。例如,大家所熟悉的排序问题,就有交换排序、选择排序、插入排序等多种算法。进行算法评价的目的,在于从解决同一个问题的不同算法中选择出较为合适的一种,同时从不同算法的比较中知道如何对现有的算法进行改进,从而设计出更好的算法。

一般从五个方面对算法进行评价。

### 一、正确性

正确性 (correctness) 是设计和评价一个算法的首要条件,如果一个算法不正确,其他方面就无从谈起。一个正确的算法是指在合理的数据输入下,能在有限的运行时间内得出正确的结果。通过对数据输入的所有可能情况的分析和上机调试可以证明算法是否正确,或找出算法的反例说明算法存在缺陷。当然,要从理论上证明一个算法的正确性更好,并不是一件容易的事情,由于本书研究的是数据结构,因此不做讨论。

### 二、健壮性

健壮性 (robustness) 是指一个算法对不合理 (又称不正确、非法、错误等) 数据输入的反应和处理能力。一个好的算法应该能够识别出错误数据并进行相应处理。对错误数据的处理一般包括打印出错信息、调用错误处理程序、返回标识错误的特定信息、中止程序运行等方式。

### 三、可读性

可读性 (readability) 是指一个算法供人们阅读和理解的容易程度。一个可读性好的算法,应该使用便于识别和记忆的、与描述事物或实现的功能相一致的标识符,应该符合结构化和模块化的程序设计思想,应该对其中的每个功能模块以及重要的数据、数据类型和语句等加以注释,应该建立有相应的文档,用来对整个算法的功能、结构、使用及有关事项进行必要的说明。

### 四、时间复杂度

时间复杂度 (time complexity) 又称计算复杂度 (computational complexity), 它是算法有效性的量度之一, 量度算法有效性的另一个重要指标是空间复杂度。时间复杂度是一个算法运行时间的相对量度。一个算法的运行时间是指在计算机上从开始到结束运行所花费的时间, 它大致等于计算机执行一种简单操作 (如赋值、转向、比较、输入、输出等) 所需时间与算法中进行简单操作次数的乘积。因为执行一种简单操作所需的时间随机器而异, 它是由机器本身硬软件环境决定的, 与算法无关, 所以只讨论影响运行时间的另一个因素——算法中进行简单操作的次数。

不管一个算法是简单还是复杂, 最终都是被分解成简单操作来具体执行的, 因此, 每一个算法都对应着一定的简单操作次数。显然, 在一个算法中, 进行简单操作的次数越少, 其运行时间也就越少; 次数越多, 运行时间也越多。所以, 把算法中包含简单操作次数的多少叫做算法的时间复杂度, 它是一个相对的量度。

若解决一个问题的规模为  $n$ , 如排序问题中,  $n$  表示待排元素的个数; 在图的遍历中,  $n$  表示图中的顶点数。那么, 算法的时间复杂度就是  $n$  的一个函数, 通常记为  $T(n)$ 。下面我们通过一些具体实例来分析算法的时间复杂度。

例题 1-4 对一个一维的实数型数组, 求它的累加和的算法。

```

Program sumA;
Const n = 100;
Var
    A: array [1..n] of real;
    i: integer;
    sum: real;
Begin
    (1) Sum := 0;
    (2) For i := 1 to n do
        Sum := sum + a[i];
    (3) Writeln ('sum = ', sum);

```

End.

计算机执行算法时, 第 (1) 步和第 (3) 步是一次赋值操作和一次输出操作, 重点是分析第 (2) 步包含多少简单操作的次数。为了便于分析, 将 (2) 改写为如下形式:

```

(2)  i := 1;                1 次
      1: if i > n then goto 2;    n+1 次
      Sum := sum + a[i];        n 次
      i := i + 1;              n 次
      Goto 1;                  n 次
(3) 2: writeln ('sum = ', sum);

```

把第 (2) 步分解后的每一条语句的执行次数加起来, 就得到了它包含的简单操作的次数, 即为  $4n+2$ 。因此, 这个求累加和算法的时间复杂度为:

$$T(n) = 4n + 4$$

例题 1-5 对一个一维的整数数组, 按从大到小排序。

```

Program sortA;
Const
    n = 10;
var
    A: array [1..n] of integer;
    i, j, p, t: integer;
Begin
    ( I ) For i := 1 to n do
        Readln (a[i]);
    ( II ) For i := 1 to n-1 do
        begin
            p := i;
            For j := i to n do
                if a[j] > a[p] then p := j;

```

```

    t := a[i]; a[i] := a[p]; a[p] := t;
end;
(Ⅲ) for i := 1 to n do
    if i mod 5 = 0 then writeln (a[i]; 5) else write (a[i]; 5);
End.

```

显然 (Ⅰ) 和 (Ⅲ) 部分执行简单操作的次数是相同的, 与例 1-4 中循环一样, 都是  $4n+2$  次。对于 (Ⅱ) 部分我们以下程序段分析:

i := 1;	1 次
1: if i ≥ n then goto 4;	n-1 次
P := i;	n-1 次
J := i;	n-1 次
2: if j > n then goto 3;	n(n-1)/2 次
if a[j] > a[p] then p := j;	n(n-1)/2 次
j := j + 1;	n(n-1)/2 次
goto 2;	n(n-1)/2 次
t := a[i];	n-1 次
a[i] := a[p];	n-1 次
a[p] := t;	n-1 次
3: i := i + 1;	n-1 次
goto 1;	n-1 次

(Ⅲ) 4: for i := 1 to n do

则得到 (Ⅱ) 执行简单操作的次数为  $2n^2 + 6n - 7$ , 因此, 这个排序算法的时间复杂度为:

$$T(n) = 2n^2 + 14n - 3$$

例题 1-4 和例题 1-5 的算法时间复杂度还比较容易计算, 因为算法比较简单, 同时 For 循环中的循环次数是固定的; 但是, 当算法较复杂, 同时包含 While 循环或 repeat/until 循环, 其时间复杂度的计算就相当困难了。实际上, 一般没有必要精确地计算出算法的时间复杂度, 只要大致计算出相应的数量级 (Order) 即可。下面我们讨论算法时间复杂度  $T(n)$  的数量级表示。

设  $T(n)$  的一个辅助函数为  $f(n)$ , 定义为当  $n$  大于等于某一足够大的正整数  $m$  时, 存在两个常数  $a$  和  $b$  ( $a < b$ ), 使得  $a < \frac{T(n)}{f(n)} < b$  均成立, 则称  $f(n)$  是  $T(n)$  的同数量级函数。把  $T(n)$  表示成数量级的形式为:

$$T(n) = O(f(n))$$

其中大写字母  $O$  为英文 Order (即数量级) 一词的第一个字母。这种表示的意思是指  $f(n)$  同  $T(n)$  只相差一个常数倍。

例如, 在例题 1-4 中, 当  $n \geq 3$  时,  $1 < \frac{T(n)}{n} < 6$  均成立, 则  $f(n) = n$ , 在例题 1-5 中, 当  $n \geq 2$  时,  $1 < \frac{T(n)}{n^2} < 9$  均成立, 则  $f(n) = n^2$ 。由此可以推出, 当  $T(n)$  是多项式时,  $f(n)$  则为  $T(n)$  的最高次幂, 若把例题 1-4 和例题 1-5 的算法时间复杂度分别用数量级的形式表

示,则为  $O(n)$  和  $O(n^2)$ 。

算法的时间复杂度采用数量级表示后,将给出一个算法的时间复杂度带来很大的方便,这时只需要分析影响一个算法运行时间的主要部分即可,不必对每一步都进行详细的分析;同时对主要部分的分析也可以简化,一般只需要分析清楚循环体内简单操作次数即可,如对于例题 1-5 我们就只需要分析 (II) 这个部分,弄清楚是一个双重循环的操作执行的次数大致为  $n^2$ ,就可得出算法的时间复杂度为  $O(n^2)$ 。

算法的时间复杂度通常具有  $O(1)$ 、 $O(n)$ 、 $O(\log_2 n)$ 、 $O(n \log_2 n)$ 、 $O(n^2)$ 、 $O(n^3)$ 、 $O(2^n)$  和  $O(n!)$  等形式。 $O(1)$  表示算法的运行时间为常量,它不随数据量  $n$  的改变而改变。如访问表中的第一个元素时,无论表的大小如何,其时间复杂度都为  $O(1)$ 。具有  $O(n)$  数量级的算法被称为线性算法,其运行时间与  $n$  成正比,如对一个表进行顺序查找时,其时间复杂度就是  $O(n)$ ;有些算法的时间复杂度为  $O(\log_2 n)$ ,即与  $n$  的对数成正比,如在有序表上进行二分查找的算法就是如此;对数组进行排序的各种简单算法的时间复杂度为  $O(n^2)$  数量级的,当  $n$  加倍时,其运行时间将增长 4 倍;对数组进行排序的各种改进算法的时间复杂度为  $O(n \log_2 n)$  数量级的,当  $n$  加倍时,其运行时间只是原来的  $2(1 + 1/\log_2 n)$  倍;做两个阶矩阵的乘法运算时,其时间复杂度为  $O(n^3)$ ;求具有  $n$  个元素集合的所有子集的算法,其时间复杂度应为  $O(2^n)$ ,因为对于含有  $n$  个元素的集合来说共有  $2^n$  个不同的子集;求具有  $n$  个元素的全排列的算法的时间复杂度为  $O(n!)$ ,因为它共含有  $n!$  种不同的排列。

随着  $n$  值的增大,各种时间复杂度的数量级的值,其增长速度是大不相同的。对数的值增长得最慢,线性值较之快些,其余依次为线性与对数的乘积、平方、立方、指数与阶乘,即阶乘的增长速度最快。因此,当  $n$  大于一定的值后,各种不同的数量级对应的值满足如下关系:

$$O(1) < O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

一个算法的时间复杂度还可具体分为最好、最差和平均三种情况。下面结合从一维数组  $A(1..n)$  中顺序查找其值等于给定值  $X$  的元素的算法进行说明。

Procedure search ( $A, n, x, k$ );

{ $K$  为一变量形参,当查找成功时,返回对应的下标,查找失败时,返回 0 值}

Begin

(1)  $i := 1$ ;

(2) while ( $i \leq n$ ) and ( $a[i] < > x$ ) do

$i := i + 1$ ;

(3) if  $i > n$  then  $k := 0$  else  $k := i$ ;

End;

此算法的时间复杂度取决于第 (2) 步的比较次数 (即循环次数加 1),而比较次数不是固定的,它与具体输入数据有关,最好的情况就是元素  $A[1]$  等于给定值  $X$ ,只要进行一次比较,显然是时间复杂度  $O(1)$ ;最坏的情况就是,没有任何元素等于给定值  $X$ ,查找失败,需要进行  $n+1$  次比较,显然时间复杂度  $O(n)$ ;当考虑到数组  $A$  中每个元素都有相同的概率 (即为  $\frac{1}{n+1}$ ) 等于给定值  $X$  时,则所需要比较的平均次数为:

$$\frac{1}{n+1} \sum_{i=1}^{n+1} i = \frac{n}{2} + 1$$

所以,从平均情况来看,算法的时间复杂度为  $O(n)$ 。

在一个算法中,最好情况的时间复杂度最容易求出,但它通常没有多大的实际意义,因为数据一般都是随意分布的,出现最好情况分布的概率最小;最差情况的时间复杂度也容易求出,它比最好情况有实际意义,通过它可以估计到算法运行时所需要的相对最长时间,并且能够使用户知道如何设计数据的排列次序,尽量避免或减少最差情况的发生;平均情况的时间复杂度最有实际意义,它确切地反映了运行一个算法的平均快慢程度,所以通常用它来表示一个算法的时间复杂度。对于一个算法来说,平均和最差这两种情况下的时间复杂度的数量级形式往往相同,它们主要差别在最高次幂的系数上;另外,有些算法最好、最差和平均的时间复杂度是相同的,如对于例 1-4 和例 1-5 就是如此。

### 五、空间复杂度

空间复杂度 (space complexity) 是对一个算法在运行过程中临时占用存储空间大小的量度,它也是衡量一个算法有效性的一个方面。一个算法在计算机存储器上所占用的存储空间,包括存储算法本身所占用的存储空间、算法的输入输出数据所占用的存储空间和算法在运行过程中临时占用的存储空间这三个方面。

算法的输入输出数据占用的存储空间是由要解决的问题所决定的,它不随算法的不同而改变。

存储算法本身所占用的存储空间与算法的书写长度成正比,要压缩这方面的存储空间,就必须编写出较短的算法,如编写成递归算法通常就比非递归算法要短。

算法在运行过程中临时占用的存储空间随算法的不同而异,有的算法只需要占用少量的临时工作单元,而且不随问题规模的大小而改变;有的算法需要占用的临时工作单元数随着问题规模  $n$  的增大而增大,当  $n$  较大时,将占用较多的存储单元,浪费存储空间,如以后要介绍的归并排序算法。

分析一个算法所占用的存储空间要从各方面综合考虑,如对于递归算法来说,一般都比较简短,算法本身所占用的存储空间较小,但运行时需要一个附加堆栈,从而占用较多的临时工作单元;若写成非递归算法,一般可能较长,算法本身所占用的存储空间较多,但运行时临时工作单元少。

一个算法的空间复杂度通常只是考虑在运行过程中为局部变量分配的存储空间的大小,它包括为参数表中形参变量分配的存储空间和为在函数体中定义的局部变量分配的存储空间两个部分。算法的空间复杂度一般也以数量级形式给出。如当一个算法的空间复杂度为一个常量,即不随被处理数据量  $n$  的大小而改变时,则表示为  $O(1)$ ; 当一个算法的空间复杂度与以 2 为底的  $n$  的对数成正比时,则表示为  $O(\log_2 n)$ ; 当一个算法的空间复杂度与  $n$  成线性比例关系时,则表示为  $O(n)$ 。对于例题 1-4,空间复杂度显然就是  $O(n)$ 。

上面讨论了如何从五个方面评价一个算法,但它们不是孤立的,而是相互联系的,除了算法的正确性外,其他几个方面往往又是相互矛盾的。如追求较短的运行时间,可能带来占用较多的存储空间;当追求占用较少的存储空间时,可能会带来较长的运行时间;当追求算法的健壮性和可读性时,可能带来占用较长的运行时间和较多的存储空间。所以在设计一个算法时,要从这几个方面综合考虑,同时还要考虑的编程复杂性。只有认真细致地从算法的这几个方面去考虑,才能设计出较好的算法。当然,在设计算法方面,还要考虑算法处理数据量的大小、算法描述语言的特性、算法运行系统的软硬件环境等因素的制约。

## 1.4 Pascal 语言中的数据类型

在计算机领域,数据类型与每一种计算机语言都有相关的概念。一般地说,计算机语言不同,对数据进行分类的规则也不同,因而产生的数据类型也不完全相同。在 Turbo Pascal 语言中,数据被划分为三大类型:简单类型、构造类型和指针类型。

### 一、简单类型

简单类型又包括整型 (integer)、实型 (real)、字符型 (char)、布尔型 (boolean)、枚举类型和子界类型,其中前四种类型是标准类型,即由系统定义的,后两种类型由用户自己定义。每一种简单类型都定义了一个具有相同数据特征的值的集合,如整型定义的值的集合是  $[-32768..32767]$  之间的所有整数,布尔类型定义的值的集合是  $[false, true]$ ,这是两个表示逻辑假和逻辑真的标识符。每个简单类型的数据都是一个无法再分割的数据“原子”,如整型数据 128、实型 3.628、字符“A”、逻辑值 true 等都是这样的数据“原子”。存储一个简单类型的数据需要占用一个存储单元,该存储单元所包含的字节数(每个字节由八位二进制位组成)由数据的类型而决定。

### 二、构造类型

构造类型又包括数组、记录、集合和文件这四种类型。每一种构造类型中的数据都由若干成分所组成,如数组(即数组类型中的数据)是由固定数目的数据元素所组成的。每一种构造类型中的数据都对应一定的逻辑结构和存储结构,下面分别进行简单的讨论。

#### 1. 数组

数组类型定义为:  $\text{array } [T1] \text{ of } T2;$

其中  $T1$  表示下标类型,它可以是除整型和实型外的其他任何简单类型,假定用  $\text{sum}(T1)$  表示  $T1$  中所含不同值的个数,则该数组类型中的每个数组均含有  $\text{sum}(T1)$  个元素。 $T2$  表示成分(即元素)类型,它可以是任何类型。

数组中的元素在位置上是顺序排列的,即第  $i$  个元素排列在第  $i-1$  个元素的后面和第  $i+1$  个元素的前面。这样元素之间在位置上的排列关系就是一种线性关系,所以数组的逻辑结构是一种线性结构,可用二元组表示为:  $\text{array} = (A, R)$

其中:

$$A = \{a_i \mid 1 \leq i \leq n, n = \text{sum}(T1), a_i \in T2\}$$

$$R = \{r\}$$

$$R = \{ \langle a_i, a_{i+1} \rangle \mid 1 \leq i \leq n-1 \}$$

对应的图形如图 1-5 所示:

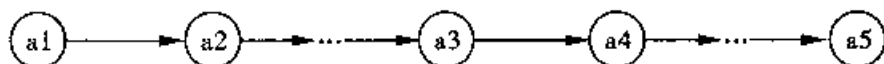


图 1-5 数组的逻辑存储结构示意图

数组的存储结构是一种顺序存储结构,即在存储空间上,数组的第  $i+1$  个元素紧挨着存储在 第  $i$  个元素的存储位置的后面。这样,数组元素之间的线性关系通过顺序存储的方式很自然地反映



出来。数组的存储结构可用图 1-6 表示。

存储地址	b	b+1*k	...	b+(i-1)*k	b+i*k	...	b+(n-1)*k
存储空间	A <sub>1</sub>	A <sub>2</sub>	...	A <sub>i</sub>	A <sub>i+1</sub>	...	A <sub>n</sub>
元素序号	1	2	...	i	i+1	...	n

图 1-6 数组的存储结构示意图

由于数组中的每个元素都具有相同的类型 T2，所以在存储空间上都占有相同的字节数（假定为 k）。若第 i 个元素 a<sub>i</sub> 的存储地址（即对应的 k 个字节中的第一个字节的地址）用 Loc(a<sub>i</sub>) 来表示，则第 i+1 个元素 a<sub>i+1</sub> 的存储地址为：

$$\text{Loc}(a_{i+1}) = \text{Loc}(a_i) + K$$

多维数组在第五章介绍。

## 2. 记录

记录类型定义的一般格式为：

Record

<域名 1> : <类型 1>;

<域名 2> : <类型 2>;

.....

<域名 n> : <类型 n>;

End;

其中 <类型 1> 至 <类型 n> 均可以是任何类型。

记录的成分（即域值）在位置上是顺序排列的，在存储空间上是顺序存放的，因此，记录同数组一样，其逻辑结构是线性结构，存储结构是顺序结构。记录同数组的区别为：

(1) 数组中的成分是一类型，而记录中的成分则允许具有不同类型；

(2) 数组中的每个成分占有的存储单元具有相同的字节数，而记录中的每个成分占有的存储单元，由于类型不同可能具有不同的字节数；

(3) 数组中的每个成分是通过下标来指明和访问的，而记录中的每个成分是通过域名来指明和访问的。

## 3. 集合

集合的类型定义为：

Set of <基类型>;

其中 <基类型> 是除整型和实型外的任何简单类型。集合类型的每一个值是一个集合。如集合类型：set of 1..3;

所包含的全部值是下列八个集合：

{ }, [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]

一个集合只与所含的元素有关，而与元素的位置无关，如 [1, 2, 3] 和 [3, 1, 2] 表示同一集合。因此，集合的逻辑结构是关系为空（即 r = {}）的结构，即元素之间不存在次序关系。

集合的存储是用一个与基类型值的个数等长的二进制串来表示的，其中每个基类型的值与一个二进制位相对应。当一个集合中包含某个基值时，则对应的二进制位用“1”表示，否则用

“0”表示。例如，有一个集合 S，其类型说明为：

S: set of 'a' .. 'f' ;

若 S 的值为 [ 'a', 'c', 'd', 'f' ]，则 S 的存储结构如图 1-7 所示：

二进制位序号	0	1	2	3	4	5
二进制位值	1	0	1	1	0	1
对应的基类型值	'a'	'b'	'c'	'd'	'e'	'f'

图 1-7 存储结构示意图

因此，集合的存储结构是顺序存储结构，即是以二进制位为单位，按照基类型值的序号（即图中的二进制位序号）顺序进行存储的结构。

集合同数组和记录相比，不同在于：

(1) 集合的元素个数是不固定的，因为它可以是基类型值所组成的任何子集（包括空集），而数组和记录中的成分个数是固定的；

(2) 集合中的元素在位置上是无序的，而数组和记录中的成分在位置上是有顺序的；

(3) 集合中的元素不能单独地访问和运算，而数组和记录中的成分都可以单独地访问和参加运算。单从这一点看，集合具有简单类型数据的特性，即只能作为整体进行访问和运算。

#### 4. 文件

文件类型的定义为：

File of < 成分类型 > ;

文件的成分（一般为记录）在位置上是顺序排列的，所以单从这一点说，文件的逻辑结构是线性结构。文件通常是被存储在外存的，其存储结构在 Turbo Pascal 7.0 上介绍了，在信息学竞赛中原则上使用的是文本文件，且我们一般也不需要定义类型，直接使用标准输入（Input）输出（Output）即可，在编程时使用下面具体范例形式：

Const

Inf = 'G1. In'; } 输入文件名 }

Out = 'G1. Out'; } 输出文件名 }

Procedure init;

Begin

Assign (Input, Inf); ReSet (Input);

ReadLn (读取数据);

...

Close (Input); } 关闭文件 }

End;

Procedure out;

Begin

Assign (Output, Out); ReWrite (Output);

WriteLn (写入数据);





...

Close (Output); | 关闭文件 |

End;

### 三、指针类型

指针类型是以存储单元的地址作为其值的一种数据类型。指针类型的定义为：

$\wedge$  < 类型标识符 >; (或  $\uparrow$  < 类型标识符 >)

如：P 为指向 integer 的指针变量，则 P 的类型说明为：

P:  $\wedge$ integer; (或 P:  $\uparrow$ typep)

那么 P (或 P $\uparrow$ ) 就表示 integer 类型的一个动态变量，P 的值就是该动态变量所对应的存储单元的首地址。

同简单变量一样，指针变量也是一种整体变量 (即变量中不包含任何成分，不可再分)，系统为每一个指针变量分配一个固定的存储单元，一般为两个字节。指针变量经常作为记录的成分，用来实现数据的动态链接存储。例如：

```
Pointer = record
    data: integer;
    next:  $\wedge$ pointer;
end;
```

其中 next 为指向 pointer 类型的指针变量，通过它把具有 pointer 类型的结点动态链接起来，从而实现对 data 域中数据的链接存储。

本书将利用 Turbo Pascal 语言所提供的各种数据类型来描述数据的各种存储结构。按照数据的存储结构就能够把数据有组织地存储到计算机中，从而方便地实现对数据的各种运算。

## 1.5 小结

本章主要介绍数据结构的一些概念和阅读本书需要的知识。

1. 数据结构研究的是数据的表示与数据之间的关系。从逻辑上讲，数据有线性结构、树结构、图结构和离散结构 (或集合结构)。从物理上讲，数据有顺序结构、链接结构、索引结构和散列结构四种。理论上，任一种逻辑结构都可以用任一种存储结构来实现。

2. 在线性结构中，数据之间是一对一的关系。在树结构中，数据之间是一对多的关系。在图结构中，数据之间是多对多的关系。在离散结构中，不考虑数据之间的任何次序，它们处于无序的、各自独立的状态。

3. 一个数组占有一块连续的存储空间，每个元素的物理存储单元是按下标位置从 0 开始连续编号的，相邻元素之间其存储位置也是相邻的。对于任一种数据的逻辑结构，若能够把元素之间的逻辑关系对应地转换为数组下标位置之间的物理关系，则就能够利用数组来实现其顺序存储结构。

4. 抽象数据类型是数据和对数据进行各种操作的集合体。这里所说的数据是广义的，是带有结构的数据，它可以具有任何逻辑结构和存储结构。



5. 算法的评价指标主要有正确性、健壮性、可读性、时间复杂度和空间复杂度。而时间复杂度和空间复杂度又往往相互矛盾, 一个算法的时间复杂度和空间复杂度越高, 越节省时间和空间。

6. 算法的时间复杂度和空间复杂度通常用数量级形式表示出来。数量级的形式分为常量级、对数级、线性级、线性乘对数级、平方级、立方级、指数级、阶乘级等多个级别; 当数据处理量较大时, 处于前面级别的算法比处于后面级别的算法更有效。

## 习题一

### 一、选择题

1. 已知下列四个算法的时间复杂度, 哪一个算法在  $[1, 100]$  中是最优的? ( )  
A.  $N!$                       B.  $1000N^2$                       C.  $N^3 + 1000$                       D.  $N + 10^{12}$
2. 已知某一算法的时间复杂度上限函数满足递归关系  $T(n) = 2(T/2) + n$ , 那么该算法的渐进时间复杂度为( )。  
A.  $O(\log_2^2 n)$                       B.  $O(n^2)$                       C.  $O(n)$                       D.  $O(n \log_2 n)$
3. 设某数据结构的任一个元素的前趋与后继元素的个数分别为  $X$  和  $Y$ , 则  $X:Y$  的值为( )  
时, 此数据结构肯定为线性结构。  
A. 1:2                      B. 1:1                      C. 2:3                      D. 3:4
4. 已知下列算法的时间渐进复杂度, 不属于多项式算法的是( )。  
A.  $O(n^{100})$                       B.  $O(2^N)$                       C.  $O(n^2)$                       D.  $O(2^{100}n)$
5. 对于某算法的时间复杂度上界函数满足递归关系  $T(n) = n + T(n-1)$ , 那么该算法的渐进复杂度为( )。  
A.  $O(1)$                       B.  $O(n)$                       C.  $O(2n)$                       D.  $O(n^2)$
6. 执行下面 Pascal 程序段的时,  $S$  被执行的次数为( )。  
程序:  
    for  $i := 1$  to  $n$  do  
        for  $j := 1$  to  $i$  do  $S$ ;

- A.  $n^2$                       B.  $n^2/2$                       C.  $n(n+1)$                       D.  $n(n+1)/2$

7. 下面 Pascal 程序 Work 执行的次数是( )。

程序:

```
for i := 1 to n do
    for j := 1 to n do
        if (i + j) mod 2 = 0 then Work;
```

- A.  $n^2$                       B.  $n(n+1)$                       C.  $n(n+1)/2$                       D.  $(n^2 - 1) \div 2 + 1$
8. 已知某一个 pascal 函数  $f(n)$ , 计算  $f(4)$  的时候函数  $f$  被调用的次数为( )。

程序:

```
function f(n: longint): longint;
begin
    if  $n \leq 1$  then exit(n) else exit(f(n-1) + f(n-2));
```

end;

A. 7

B. 8

C. 9

D. 10

9. 一个算法的时间复杂度为  $(3n^2 + 2n \log n + 4n - 7) / 5n$ , 则其数量级可以表示为( )。

A.  $O(n^3)$

B.  $O(n^2)$

C.  $O(n)$

D.  $O(1)$

10. 下面 Pascal 程序的时间复杂度为( )。

程序:

for i: =1 to n do

for j: =1 to n do

a[i][j]: =i\*j;

A.  $O(n)$

B.  $O(n^2)$

C.  $O(2^n)$

D.  $O(1)$

二、有下列几种用二元组表示的数据结构, 试画出它们分别对应的图形表示, 并指出它们分别属于何种结构。

1.  $A = (K, R)$ , 其中

$K = \{a, b, c, d, e, f, g, h\}$

$R = \{r\}$

$r = \{\langle a, b \rangle, \langle b, c \rangle, \langle c, d \rangle, \langle d, e \rangle, \langle e, f \rangle, \langle f, g \rangle, \langle g, h \rangle\}$

2.  $B = (K, R)$ , 其中

$K = \{a, b, c, d, e, f, g, h\}$

$R = \{r\}$

$r = \{\langle d, b \rangle, \langle d, g \rangle, \langle b, a \rangle, \langle b, c \rangle, \langle g, e \rangle, \langle g, h \rangle, \langle e, f \rangle\}$

3.  $C = (K, R)$ , 其中

$K = \{1, 2, 3, 4, 5, 6\}$

$R = \{r\}$

$r = \{(1, 2), (2, 3), (2, 4), (3, 4), (3, 5), (3, 6), (4, 5), (4, 6)\}$

4.  $D = (K, R)$ , 其中

$K = \{48, 25, 64, 57, 82, 36, 75\}$

$R = \{r_1, r_2\}$

$r_1 = \{\langle 25, 36 \rangle, \langle 36, 48 \rangle, \langle 48, 57 \rangle, \langle 57, 64 \rangle, \langle 64, 75 \rangle, \langle 75, 82 \rangle\}$

$r_2 = \{\langle 48, 25 \rangle, \langle 48, 64 \rangle, \langle 64, 57 \rangle, \langle 64, 82 \rangle, \langle 25, 36 \rangle, \langle 82, 75 \rangle\}$

三、用类 Pascal 语言描述下列每一个算法, 并分别求出它们的时间复杂性。

1. 求一维实型数组  $A(1:n)$  中的所有元素之乘积。

2. 计算  $\sum_{i=1}^n \frac{x_i}{i+1}$  的值。

3. 假定以为整型数组  $A(1:n)$  中的每一个元素均在  $[0, 200]$  区间内, 分别统计处在  $[0, 20)$ 、 $[20, 50)$ 、 $[50, 80)$ 、 $[80, 130)$ 、 $[130, 200]$  等各区间内的元素个数。

四、指出下列各算法的功能并求出其时间复杂性。

1. procedure prime(n);

begin

(1) i: =2;

```

(2) while ( (n mod i) < > 0) and (i < sqrt (n)) do i: = i + 1;
(3) if i > sqrt (n) then writeln (n: 6, 'is a prime number')
    else writeln (n: 6, 'is not a prime number')
end;
2. function sum1 (n): real; {n 的值为一个正整数}
begin
    (1) p: = 1; sum: = 0;
    (2) for i: = 1 to n do
        [p: = p * i; sum: = sum + p];
    (3) sum1: = sum
end;
3. function sum2 (n): real; {n 的值为一个正整数}
begin
    (1) sum: = 0;
    (2) for i: = 1 to n do
        (i) p: = 1;
        (ii) for j: = 1 to i do p: = p * j;
        (iii) sum: = sum + p
    (3) sum2: = sum
end;
4. procedure sort (A, n);
begin
    for i: = 1 to n - 1 do
        (i) k: = i;
        (ii) for j: = i + 1 to n do
            if a [j] < a [k] then k: = j;
        (iii) if k < > i then a [i]  $\leftarrow$  a [k]
end;
5. procedure matmult (A, B, C);
    {A 为 m * n 阶矩阵, B 为 n * l 阶矩阵, C 为 m * l 阶矩阵}
begin
    (1) for i: = 1 to m do
        for j: = 1 to l do c [i, j]: = 0;
    (2) for i: = 1 to m do
        for j: = 1 to l do
            for k: = 1 to n do
                C [i, j]: = C [i, j] + A [i, k] * B [k, j]
end;

```

## 2 线性表

线性表是最简单、最基本、最常用的一种线性结构。它有两种存储的方法：顺序存储和链式存储。它的基本操作是插入、删除和检索等。

### 2.1 线性表的定义和顺序存储

#### 一、线性表的定义

线性表 (linear list) 是具有相同特性的数据元素的一个有限序列。该序列中所含元素的个数叫做线性表的长度, 用  $n$  表示,  $n \geq 0$ 。当  $n = 0$  时, 表示线性表是一个空表, 即表中不包含任何元素。设序列中第  $i$  个元素为  $a_i$  ( $1 \leq i \leq n$ ), 则线性表的一般表示为:

$$(a_1, a_2, a_3, \dots, a_i, \dots, a_n)$$

其中  $a_1$  为第一个元素, 又称作表头元素,  $a_n$  为最后一个元素, 又称作表尾元素。一个线性表可以用一个标识符来命名, 如用  $A$  命名上面的线性表, 则

$$A = (a_1, a_2, a_3, \dots, a_i, \dots, a_n)$$

线性表中的元素在位置上是有顺序的, 即第  $i$  个元素  $a_i$  处在第  $i-1$  个元素  $a_{i-1}$  的后面和第  $i+1$  个元素  $a_{i+1}$  的前面, 这种位置上的有序性就是一种线性关系, 所以线性表是一种线性结构, 用二元组表示为:

$$\text{Linear\_list} = (A, R),$$

其中

$$A = \{a_i \mid 1 \leq i \leq n, n \geq 0, a_i \in \text{elemtype}\}$$

$$R = \{r\}$$

$$r = \{ \langle a_i, a_{i+1} \rangle \mid 1 \leq i \leq n-1 \}$$

对应的逻辑图如图 2-1 所示:

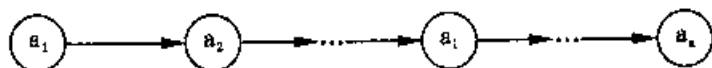


图 2-1 线性表的逻辑结构示意图

其中元素类型  $\text{elemtype}$  可表示任何一种类型。

由线性表的定义可知, 线性表的长度是可变的。当向线性表中插入一个元素时, 其长度就增加 1; 当从线性表中删除一个元素时, 其长度就减少 1。

线性表是一种线性结构, 反过来, 任何线性数据结构都可以用线性表的形式表示出来, 这只是按照元素之间的逻辑关系把它们顺序排列即可。

## 二、线性表的顺序存储

线性表的顺序存储是线性表的一种最简单的存储结构，其存储方式是：在内存中为线性表开辟一块连续的存储空间，该存储空间所包含的存储单元数要大于等于线性表的长度（假定每个存储单元具有存储线性表中一个元素所需要的存储字节数），让线性表的第一个元素存储在这个存储空间的第一个单元中，第二个元素存储在第二个单元中，依此类推。

因为一个数组在内存中对应着一块连续的存储空间，所以我们可以借用数组来为线性表的顺序存储开辟存储空间，该数组的单元数（即成分个数）要大于等于线性表的长度。另外，为了存储线性表的长度，还要使用一个整型变量。若把线性表的顺序存储所使用的一个数组和一个整型变量统一说明在一个记录类型中，则该记录类型可定义为：

```
type list = record
    vec: array [1.. m0] of elemtype;
    len: integer;
end;
```

其中  $vec$  域（即一维数组）用来顺序存储线性表中的所有元素， $len$  域（即整型变量）用来存储线性表的长度，亦即表尾元素所对应的下标， $m_0$  表示在线性表动态变化的过程中可能需要的最大长度，其值由用户决定。

对于上面线性表  $A$  来说，它的顺序存储结构如图 2-2 所示。

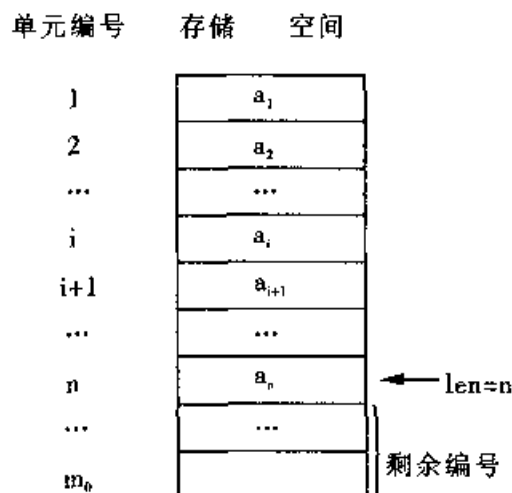


图 2-2 线性表的顺序存储结构示意图

## 2.2 线性表的运算

对线性表的基本运算有：

1. 置一个线性表为空表；
2. 求线性表的长度；
3. 读取线性表中第  $i$  个元素；
4. 修改或者重写线性表中第  $i$  个元素；

5. 查找线性表中首先满足给定条件的元素;
6. 统计、累加或打印线性表中满足给定条件的所有元素;
7. 向线性表中第  $i$  个元素或满足给定条件的第一个元素所在位置插入一个新元素;
8. 删除线性表中第  $i$  个元素或满足给定条件的第一个元素。

在上面的 8 种基本运算中, 若按照是否改变线性表长度来划分, 则第 1、7、8 种运算将改变线性表的长度, 其余运算均不改变线性表的长度。对于改变线性表长度的那几种运算来说, 第一种运算将置线性表的长度为 0, 第 7 种运算将使线性表的长度增 1, 第 8 种运算将使线性表的长度减 1。

由上面的基本运算, 可以构成其他较复杂的运算。如通过置空表的运算和反复向表尾之后插入新元素的运算, 可在计算机上建立一个线性表 (确切地说, 是建立线性表的存储结构); 通过反复执行删除第  $i$  个元素的运算, 可删除线性表中从第  $i$  元素开始的连续若干个元素。

根据线性表的运算和线性表的存储结构可写出相应的算法, 存储结构不同, 其算法也不同。在顺序存储方式下, 由于线性表是利用一维数组存储的, 所以对于那些与线性表长度无关的运算, 其算法就是对数组运算的算法, 这里就不作讨论了 (当然, 对于查找运算和排序运算, 因为它们都是数据处理领域非常重要的两种运算, 所以将在后面的章节专门讨论它们的各种算法), 在此我们只讨论与线性表长度操作有关的算法。

在下面所列举的算法中, 假定用  $L$  表示具有顺序存储结构的类型为  $\text{list}$  的一个线性表,  $x$  表示类型为  $\text{elemtype}$  的一个数据元素,  $i$  表示类型为整型的一个元素下标。  $i$  表示类型为整型的一个元素下标。

#### 1. 置线性表为一个空表

此算法很简单, 只需要把线性表的长度置 0 即可。

$\text{SetNull}(L)$

$L.\text{len} \leftarrow 0$

#### 2. 求线性表的长度

此算法也非常简单, 只要取出线性表的长度值即可。

$\text{Length}(L)$

$\text{return}(L.\text{len})$

#### 3. 向线性表中第 $i$ 个元素位置插入一个新元素

算法步骤为:

- (1) 检查线性表的存储空间是否已被占满, 若占满, 则进行“溢出”错误处理;
- (2) 检查  $i$  值是否超出所允许的范围 ( $1 \leq i \leq n+1$ ), 若超出, 则进行“超出范围”错误处理;
- (3) 将线性表的第  $i$  个元素和它后面的所有元素均后移一个位置;
- (4) 将新元素写入到空出的第  $i$  个位置上;
- (5) 使线性表的长度增 1。

算法描述为:

$\text{Insert}(L, i, x)$

$\text{if } L.\text{len} = n_0$

$\text{then error}('overflow')$

```

if (i < 1) or (i > L.len + 1)
    then error ( 'out of range' )
for j ← L.len downto i do
    L.vec [j + 1] ← L.vec [j]
L.vec [i] ← x
L.len ← L.len + 1

```

算法中的第(1)步和第(2)步可根据情况进行取舍。若你确信线性表在插入过程中不会造成溢出,则可省去第(1)步;若你确信所给的*i*值不会超出所允许的范围,则可省去第(2)步。

此算法的时间复杂性主要取决于第(3)步的循环次数(即向后移动的元素个数)。此循环次数不仅与线性表的长度*n*(即L.len值)有关,而且与*i*值有关。当*i*=*n*+1时,元素的移动次数最少,即0次;当*i*=1时,元素的移动次数最多,即*n*次;若用*p<sub>i</sub>*表示在第*i*个元素位置插入一个新元素的概率,则在长度为*n*的线性表中插入一个元素所需移动次数的期望值(即平均次数)为 $\sum_{i=1}^{n+1} p_i (n-i+1)$ 。不失一般性,我们假定在线性表的任何位置上插入元素的概率都相同,即为

$p_i = \frac{1}{n+1}, 1 \leq i \leq n+1$ , 则元素移动的平均次数为:

$$\frac{1}{n+1} \sum_{i=1}^{n+1} (n-i+1) = \frac{n}{2}$$

可见,在向顺序存储的线性表的第*i*个元素位置插入一个元素时,最好情况是不移动元素,最坏情况是移动表中的所有元素,平均情况是移动表中的一半元素,因此,在最坏和平均这两种情况下,此算法的时间复杂度均为 $O(n)$ 。

#### 4. 删除线性表中的第*i*个元素

算法步骤为:

- (1) 检查*i*值是否超出所允许的范围( $1 \leq i \leq n$ ),若是,则进行“超出范围”错误处理;
- (2) 把第*i*个元素赋给变参X带回;
- (3) 把第*i*个元素后面的所有元素前移一个位置;
- (4) 使线性表的长度减1。

算法描述为:

```

Delete (l, i, X);
if (i < 1) or (i > L.len)
    then error ( 'out of range' )
X ← L.vec [i]
for j ← i + 1 to L.len do
    L.vec [j - 1] ← L.vec [j]
L.len ← L.len - 1

```

此算法中的第(1)步和第(2)步可根据情况进行取舍。若确信*i*值不会超过范围,则可省去第(1)步;若不需要保留被删除的元素,则可省去第(2)步,当然连参数表中的X参数也省略了。

此算法的时间复杂性主要取决于第(3)步的循环次数(即元素向前移动的次数)。同插入的过程一样,此循环次数不仅与线性表的长度*n*有关,而且与*i*值有关。当*i*=1时,元素的移动次



数最多, 即  $n-1$  次; 当  $i=n$  时, 元素的移动次数最少, 即 0 次; 若考虑平均情况, 则在删除任一元素概率相等的情况下, 元素移动的平均次数为  $\frac{n-1}{2}$ 。因此在最坏和平均两种情况下, 此算法的时间复杂性均为  $O(n)$ 。

插入和删除的示例图:

例如, 有一个线性表为

$L = (3, 17, 13, 7, 103, 27, 29, 101, 89)$

当在第 3 个元素位置插入一个新元素 19 时, 插入前后所对应的存储结构如图 2-3 (a) 和 (b) 所示 (假定  $m$  取 15)。

删除第 4 个元素位置, 得到的存储结构如图 2-3 (c) 所示。

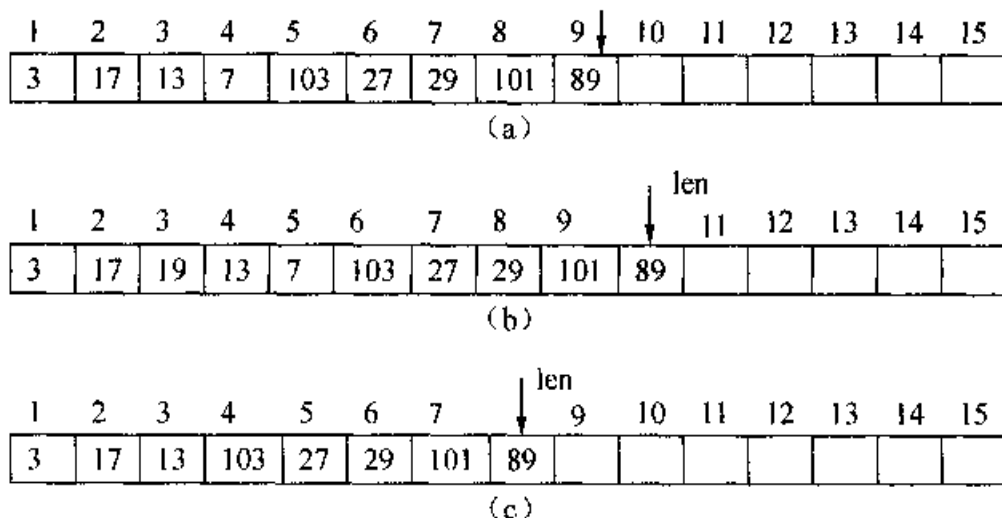


图 2-3 线性表的插入和删除示例

#### 5. 从线性表中删除其值等于给定值 $x$ 的第一个元素

算法步骤为:

- (1) 从线性表的第一个元素起, 使给定值  $x$  依次同每一个元素进行比较, 直到  $x$  等于某个元素的值 (即查找成功) 或查完所有元素仍找不到值为  $x$  的元素 (即查找失败) 为止;
- (2) 若查找失败, 则进行“没有找到”错误处理;
- (3) 删除其值等于  $x$  的那个元素;
- (4) 使线性表的长度减 1。

算法描述为:

Delete ( $L, x$ )

$i \leftarrow 1$

while ( $1 \leq L.len$ ) and ( $x \neq L.vec[i]$ ) do

$i \leftarrow i + 1$

if  $i > L.len$  then error (“not find”)

for  $j \leftarrow j + 1$  to  $L.len$  do

$L.vec[j - 1] \leftarrow L.vec[j]$

$L.len \leftarrow L.len - 1$

## 2.3 线性链表及链接存储

线性链表是线性表的链接存储表示,既可以用静态链表实现,也可以用动态链表实现。一个链接表由  $n$  个 ( $n \geq 0$ , 若  $n=0$ , 则称为空表) 结点所组成,每个结点除了包含有存储数据元素的值域外,还包含有用来实现数据元素之间逻辑关系的一个或若干个指针域,用示意图表示一个结点的结构为  $\boxed{\text{data}} \boxed{t_1} \boxed{t_2} \cdots \boxed{t_m}$ , 其中  $\text{data}$  表示值域,用来存储一个元素,  $t_1, t_2, \cdots, t_m$  ( $m \geq 1$ ) 分别表示指针域,每个指针域的值为其后继元素或前驱元素所在结点(以后简称为后继结点或前驱结点)的存储位置,若某个指针域不需要指向任何结点,则令它的值为  $\text{nil}$  (即空)。

在一个链接表中,若每个结点只包含有一个指针域(即  $m=1$  的情况),则被称为单链表,否则被称为多链表。对于线性数据结构来说,由于数据元素之间是 1:1 的联系,所以当进行链接存储时,一种最简单的方法是:在每个结点中只设置一个指针域,用以指向其后继结点,这样构成的链接表称之为线性单向链接表,简称单链表;另一种可能采用的方法是:在每个结点中设置两个指针域,分别用以指向其前驱结点和后继结点,这样构成的链接表称之为线性双向链接表,简称双向链表。若表尾结点的后继结点指向表头,表头结点的前驱结点指向表尾,整个链表就构成了一个环,被称作循环链表。

设一个线性表为:

$$A = (a_1, a_2, \cdots, a_i, a_{i+1}, \cdots, a_n)$$

若分别用单链表、双向链表和循环链表表示,则对应的示意图分别如图 2-4 (a)、(b) 和 (c) 所示。

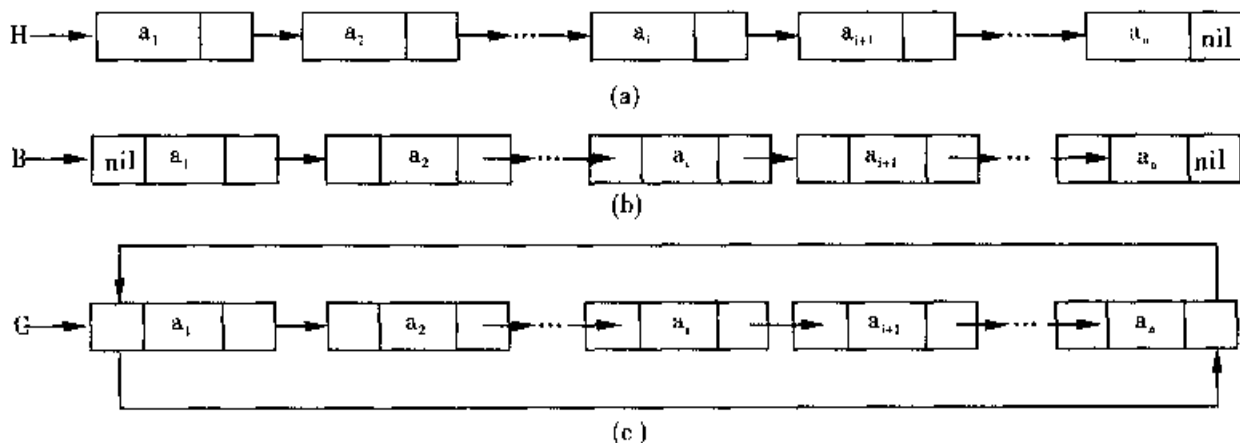


图 2-4 线性表的链接存储

其中线性表  $A$  中的第  $i$  个元素 ( $1 \leq i \leq n$ ) 存于单链表或双向链表中的第  $i$  个结点的值域。在单链表中,最后一个结点无后继结点,所以其指针域为空(用  $\text{nil}$  符号表示);在双向链表中,第一个结点无前驱结点,最后一个结点无后继结点,所以相应的指针域也为空。而循环链表的所有结点的前驱结点和后继结点都不为空。

每个链接表都由一个指针指向其第一个结点(即表头结点),我们把这个指针(如图 2-4 中的  $H$  和  $B$ ) 叫做表头指针。虽然表头指针只指向表头结点,但从表头指针出发,沿着结点的链



(即指针域的值)可以访到任一个结点。鉴于表头指针的重要性,通常以它命名一个链接表,如以 H 为表头指针的单链表,可称作 H 单链表。

在线性表的顺序存储中,逻辑上相邻的元素,其对应的存储位置也相邻,所以当进行插入或删除运算时,需要平均移动半个表的元素,这是相当费时的操作。在线性表的链接存储中,逻辑上相邻的元素,其对应的存储位置是通过指针来链接的,因而每个结点的存储位置可以任意安排,不必要求相邻,所以当进行插入或删除运算时,只需修改相关结点的指针域即可,这是既方便又省时的操作。由于链接表的每个结点带有指针域,因而在存储空间上比顺序存储要付出较大的代价。

下面通过示意图说明如何在单链表中插入和删除结点。

图 2-5 (a) 和 (b) 分别给出了在 c 结点(即存放元素 c 的结点的简称,下同)的前面插入 b 结点的前后状态。在插入过程中,首先将指向 c 结点的指针 q(即 a 结点指针域的值)赋给 b 结点的指针域,然后再将指向 b 结点的指针 p(即指针变量 s 的值)赋给 a 结点的指针域即可。

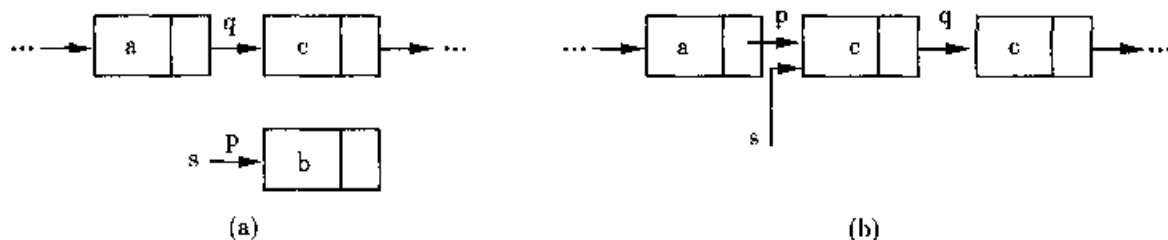


图 2-5 在单链表中插入结点的示意图

图 2-6 (a) 和 (b) 分别给出删除 y 结点的前后状态。在删除过程中,首先将指向 y 结点的指针 p(即 x 结点指针域的值)赋给一个指针变量 s,以便处理和回收该结点,然后再将指向 z 结点的指针 q(即 y 结点指针域的值)赋给 x 结点的指针域即可。

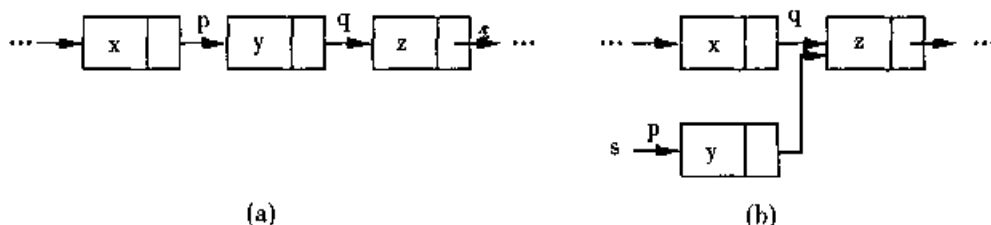


图 2-6 在单链表中删除结点的示意图

在 pascal 语言中,可采用两种方法产生存储结点:一种是通过调用 new(p) 过程,由系统分配动态结点,另一种是取自被说明的数组中的单元(即静态结点)。由动态结点链接而成的链接表被称作动态链接表,而由静态结点链接而成的链接表被称作静态链接表。对于一个单链表,若采用动态结点,则指针类型和结点类型可定义为:

```
type linklist = ↑ dynanode
    dynanode = record
        data: elemtype;
        next: linklist
    end
```

若采用静态结点,则结点类型可定义为:



```
type statinode = record
    data: elemtype;
    next: integer
end
```

在静态结点中,因 next 域用来存放其后继元素所在单元的编号(即数组下标),所以被定义为整型。

为单链表提供静态结点的数组类型可定义为:

```
type statilist = array [1.. m0] of statinode;
```

其中  $m_0$  常量表示提供给单链表的最大存储单元(即静态结点)数,其值由用户决定。

图 2-7 (a) 就是一个静态单链表的存储映像,其中 f 为表头指针 next 域的数值。0 表示空指针;图 2-7 (b) 是它的示意图,其中每个指针上标出的数值就是该指针的具体值。

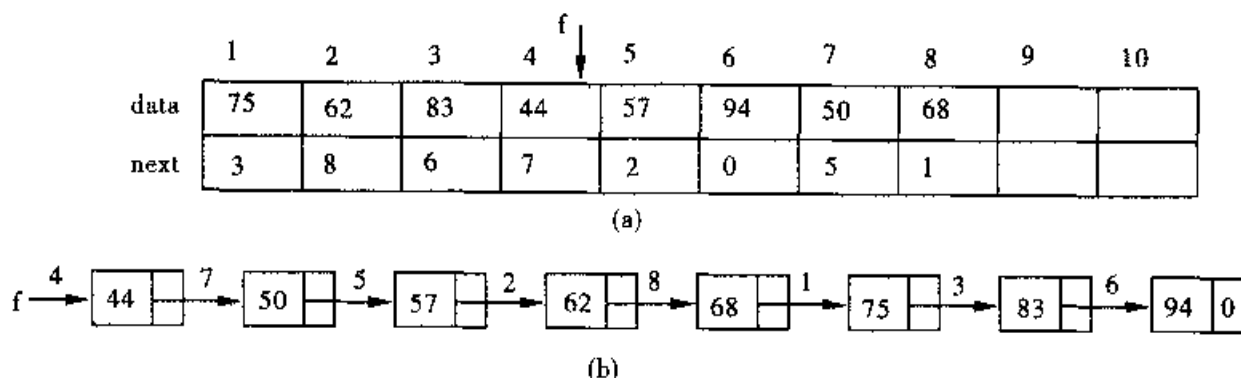


图 2-7 线性表的静态链接存储映像

从图 2-7 (b) 可以清楚地看出:该静态单链表中各数据元素之间的逻辑顺序为:

44, 50, 57, 62, 68, 75, 83, 94

在线性链接表中,有时为了运算的方便,常在第一个结点之前增设一个结点,把它称之为附加表头结点,并让链接表的表头指针指向这个结点,而这个结点的指针域则指向第一个结点。

线性链接表的基本运算和上一节所讲的线性表相同,只是其算法和相应的实现方式不同。

设 HL 为单链表的表头指针,类型为 linklist,设 x 为 elemtype 类型的参数, i, k 分别为整型参数,下面分别给出求单链表的长度,对单链表进行查找、插入、删除等常用运算的算法。

1. 求单链表的长度。

单链表的长度是指单链表中所含的结点的个数。此算法需要从表头指针 HL 出发沿着每个结点的链,依次向下访问并进行计数,直到最后一个结点为止。

算法描述为:

Length (HL)

p ← HL

j ← 0 | 分别给指针变量 p 和计数变量 j 赋初值 |

while p ≠ nil do

    j ← j + 1 (进行计数)

    p ← p ↑ . next | 修改 p 指针,使之指向下一个结点 |

return (j)

除了上面这种做法,还可以对于每个链表增加一个长度域,专门统计每个链表的长度。这个标志就如同前面的线性表所介绍,添加一个元素长度加一,删除一个元素长度减一,这里就不多介绍了。

2. 从单链表中查找出其关键字(假定为整型)等于给定值  $k$  的结点。

假定结点中带整型关键字的 `elemtype` 类型定义为:

```
type elemtype = record
    key: integer;
    .....
end
```

其中 `key` 表示关键字域,省略部分表示一些非关键字域。当然也可以把带整型关键字的结点类型定义为:

```
type dynanode = record
    key: integer;
    .....
    next: linklist
end
```

其中 `key` 域和省略部分的非关键字域合起来为结点的值域。

此算法的基本思路是:从表头第一个结点起,依次使每个结点的关键字同给定值  $k$  进行比较,直到某个结点的关键字等于给定值  $k$  (即查找成功)或者查到表尾(即查找失败)为止。

算法描述为:

```
locate (HL, k)
p ← HL (用指针 p 指向待比较的结点)
while p ≠ nil do
    if p↑.data.key = k
    then break
    else p ← p↑.next
| 若条件成立,表明查找成功,则退出循环,否则让 p 指向下一个结点,继续查找 |
return (p)
```

在这个算法中,当查找成功时,返回具有关键字  $k$  的结点的地址,否则返回 `nil`。

3. 在单链表中第  $i$  个结点 ( $i \geq 0$ ) 之后插入一个元素为  $x$  的结点。

算法步骤为:

(1) 为待插入元素  $x$  分配一个结点(假定是由  $s$  指针变量所指向的结点,即  $s \uparrow$  结点),并把  $x$  赋给  $s \uparrow$  结点的值域;

(2) 如果  $i = 0$ ,则将  $s \uparrow$  结点插入表头后返回;

(3) 从单链表中查找第  $i$  个结点;

(4) 若查找成功,则在第  $i$  个结点后插入  $s \uparrow$  结点,否则表明  $i$  值超出单链表的长度,应进行“超出范围”错误处理。

算法描述为:

```
Insert (HL, i, x)
```

```

new (s)
s↑.data ← x
if i = 0 then
    s↑.next ← HL
    HL ← s
    return
P ← HL
j ← 1 | 用指针 P 指向单链表中的第 j 个结点 |
while (p ≠ nil) and (j < i) do
    j ← j + 1
    p ← p↑.next
if p ≠ nil
then | 若条件成立, 则表明查找成功 |
    s↑.next ← p↑.next | 使 s↑ 结点的指针域指向 p↑ 结点的后继 |
    p↑.next ← s | 使 p↑ 结点的指针域指向 s↑ 结点 |
else
    error ( 'out of range' )
    
```

图 2-8 给出了在单链表中第  $i$  个结点之后插入  $x$  结点的示意图, 图中带箭头的实线为插入前的链, 虚线为插入后的链。

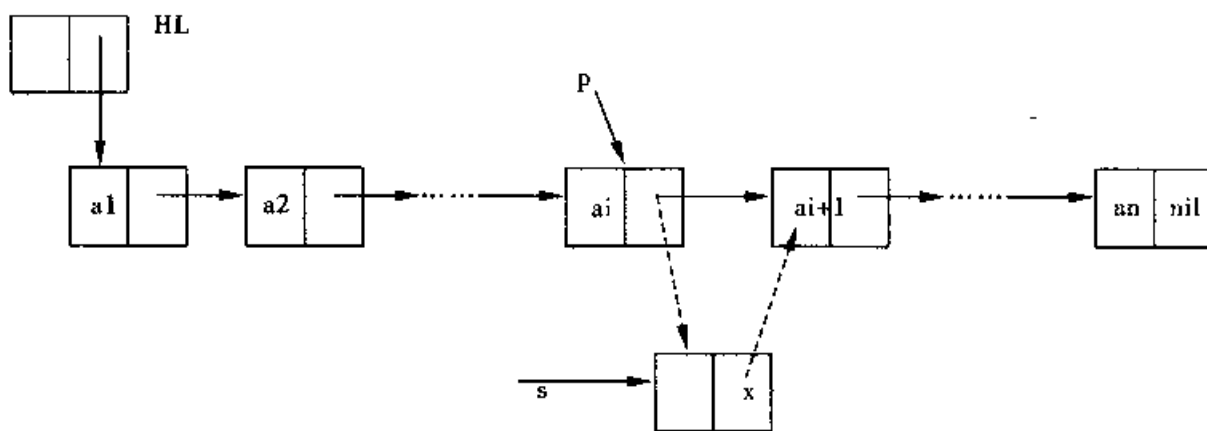


图 2-8 在第  $i$  个结点后插入新结点的链接示意图

4. 从单链表中删除其值等于给定值  $x$  的第一个结点。

算法步骤为:

- (1) 如果单链表为空, 则进行“下溢”处理;
- (2) 如果表头结点是被删除的结点, 则删除该结点后返回;
- (3) 从第二个结点起, 查找其值等于  $x$  的结点, 直到查找结束 (即查找成功或查找失败) 为止;
- (4) 若查找成功, 则删除被查找到的结点, 否则进行“没有找到”错误处理。

算法描述为:

Delete (HL,  $x$ )

```

if HL = nil
  then error ( 'underflow' )
if HL↑.data = X
  then
    p ← HL | 把表头指针赋给 p, 以便删除表头结点后回收该结点
    HL ← HL↑.next | 删除表头结点
    dispose (p) | 系统回收由 p 所指向的结点, 即原表头结点
    return | 返回
q ← HL
p ← q↑.next | p 指向待比较的结点, q 指向 P 的前驱结点
while p ≠ nil do
  if p↑.data = x
    then break
  else
    q ← p
    p ← p↑.next
if p ≠ nil
  then
    q↑.next ← p↑.next | 删除 p↑ 结点, 即值为 x 的结点
    dispose (p) | 回收 p 结点
  else
    error ( 'not found' )

```

5. 在带有附加表头结点的循环双向链表中进行插入和删除。

在双向链表中, 若采用动态结点, 则结点类型可定义为:

```

type dydunode = record
  data: elemtype;
  prev, next: dydunode
end

```

其中 data 为存储数据元素的值域, prev 为指向其前驱结点的指针域, next 为指向其后继结点的指针域。

设 p 和 q 分别是具有 ↑ dydunode 类型的指针变量, 若要在链接表中 p 所指向的结点之后插入一个 q 所指向的新结点, 则运算步骤为:

```

q↑.next ← p↑.next | 使 q 结点的后继指针指向 p 结点的后继结点
q↑.prev ← q | 使 q 结点的前驱指针指向 p 结点
p↑.next↑.prev ← q | 使 p 结点的后继结点的前驱指针指向 q 结点
p↑.next ← q | 使 p 结点的后继指针指向 q 结点

```

若要删除指针 p 所指向的结点, 则运算步骤为:

```

p↑.prev↑.next ← p↑.next | 使 p 结点的前驱结点的后继指针指向 p 的后继结点
p↑.next↑.prev ← p↑.prev | 使 p 结点的后继结点的前驱指针指向 p 的前驱结点

```

dispose (p) {回收 p 结点}

通过对上面的各种算法的分析可知,前4个算法的时间复杂性为  $O(n)$ 。对于第5个算法,若指针  $p$  已知,则插入或删除算法的时间复杂性为  $O(1)$ ;若指针  $p$  未知,需要根据条件查找连接表而确定,则算法的时间复杂性为  $O(n)$ 。

## 2.4 线性表的应用举例

**例题 2-1** 向一个有序表中插入一个新元素,使得插入后仍按关键字有序。

所谓有序表这里是指按照元素的关键字的升序(即从小到大)或降序(即从大到小)排列的线性表(关于有序表的确切含义将在后面排序一章给出)。一般不特别指明时,均把有序表看作为升序的。

假定有序表中元素的类型为:

```
elemtype = record
  key : integer;
  .....
end
```

其中包含有整型关键项  $key$  域和其他相关的域,由于不需要关心其他域的情况,所以没有给出它们的具体说明。

不难分析,完成这个任务要分成两个步骤:首先要为新元素寻找插入位置,然后再进行插入。寻找插入位置的方法很多,假定这里采用从有序表的第一个元素起,顺序查找插入位置的方法。

此题的算法步骤为:

(1) 从有序表的第一个元素起,让新元素的关键字依次同每一个元素的关键字进行比较,直到新元素的关键字小于某个元素的关键字或者比较到最后一个元素为止;

(2) 由搜索到的那个元素起,把它和后面的所有元素均后移一个位置;

(3) 把新元素写入到空出的位置上;

(4) 线性表的长度增1。

注意:在这个算法步骤中没有考虑有序表溢出的情况。

算法描述为:

Insert (L, x)

$i \leftarrow 1$

while ( $i \leq L.len$ ) and ( $x.key \geq L.vec[i].key$ ) do

$i \leftarrow i + 1$

for  $j \leftarrow L.len$  downto  $i$  do

$L.vec[j+1] \leftarrow L.vec[j]$

$L.vec[i] \leftarrow x$

$L.len \leftarrow L.len + 1$



如果用初始的有序表是一个链接表的话,那么就需要稍作改动。首先同样用枚举的方法,从第一个结点开始找到元素应该插入的位置,然后用链接表插入的方法进行插入即可。

```

Insertl (HL, x)
    new (p)
    p↑.key ← x
    q ← HL
    if x ≤ q↑.key then
        p↑.next ← q
        q↑.prev ← p
        HL ← p
    return
    while (q↑.next ≠ nil) and (q↑.next↑.key ≤ x) do
        q ← q↑.next
    if q↑.next ≠ nil then
        p↑.next ← q↑.next
        q↑.next↑.prev ← p
    q↑.next ← p
    p↑.prev ← q

```

算法1的时间复杂性主要取决于第(1)步为寻找插入位置所需的比较次数和第(2)步为空出插入位置所需的移动次数。假定新元素的插入位置为 $i$ ,则元素的比较次数为 $i$ 次,元素的移动次数为 $n-i+1$ 次,两者相加为 $n+1$ 次。也就是说,不管新元素插入在什么位置上,元素的比较和移动次数的总和不变,均为 $n+1$ ,所以此算法的时间复杂性为 $O(n)$ 。

而算法2的查找时间复杂度为 $O(n)$ ,而插入的时间复杂度仅为 $O(1)$ ,所以总的时间复杂度仍为 $O(n)$ 。

**例题2-2** 设有两个线性表LA和LB,这两个线性表都是从小到大排好序,且每个线性表中都没有相同的元素。现在要求将两个线性表合并成一个线性表LC,合并的原则是:

- (1) 合并后的新表同样也满足从小到大的顺序;
- (2) LC中没有两个相同的元素。

算法步骤:

- (1) 将LC清空;
- (2) 按照从小到大的顺序将LA和LB中元素插入LC中。

算法描述如下:

```

Merge (LA, LB)
    LC.len ← 0
    i ← 1 {i 指向 LA 当前要插入 LC 中的元素}
    j ← 1 {j 指向 LB 当前要插入 LC 中的元素}
    while (i ≤ LA.len) and (j ≤ LB.len) do

```

```

inc (LC. len)
if LA. vec [i] < LB. vec [j] then
    { 如果 LA 当前要插入 LC 中的元素比 LB 中的小, 那么先插入 LA 的 }
    LC. vec [LC. len] ← LA. vec [i]
    i ← i + 1
else
    if LA. vec [i] = LB. vec [j] then
        { 如果 LA 当前要插入 LC 中的元素和 LB 中的相同, 那么只插入一个 }
        LC. vec [LC. len] ← LA. vec [i]
        i ← i + 1
        j ← j + 1
    else
        { 如果 LA 当前要插入 LC 中的元素比 LB 中的小, 那么先插入 LB 的 }
        LC. vec [LC. len] ← LB. vec [j]
        j ← j + 1
while i ≤ LA. len do
    { 把 LA 中剩下的元素插入 LC }
    inc (LC. len)
    LC. vec [LC. len] ← LA. vec [i]
    i ← i + 1
while j ≤ LB. len do
    { 把 LB 中剩下的元素插入 LC }
    inc (LC. len)
    LC. vec [LC. len] ← LB. vec [j]
    j ← j + 1

```

因为 LA、LB 是两个有序表, 按照从小到大比对插入, 保证了不重不漏, 而且合并之后同样是有序表。由于指针  $i, j$  最多移动有序表的长度次, 因此复杂度是  $O(n)$  的。同样合并操作也可用链接表实现, 具体过程就请读者自行思考。

### 例题 2-3 约瑟夫环。

有  $n$  个人围坐在一张圆桌周围, 现从第一个人开始报数, 数到  $m$  的人出列 (即离开座位), 接着从出列的下一个人开始重新报数, 数到  $m$  的人有出列 (出列的人不再参加报数), 如此下去直到所有人都出列为止, 试求出它们的出列次序。

例如, 当  $n=8, m=4$  时, 从第一个人 (假定每个人的编号依次为 1、2、……、 $n$ ) 开始报数, 则得到的出列次序为: 4 8 5 2 1 3 7 6。

要求: 给出  $n, m$ , 求出每个人的出列顺序 H。

解析:

题目中已经把出列方法讲解得十分清楚了, 我们只需要按照选择正确的数据结构模拟出列即



可。由于只牵涉到报数和删除,选择链接表是最好的,因为删除只需要  $O(1)$  的时间复杂度。

用循环链接表存储  $n$  个结点,每个结点表示围着圆桌的人。每次报数到第  $m$  个就删除此人。(当  $m > n$  时,  $m \bmod n$  和  $m$  是等效的)

下面是伪代码的描述:

```

Josephus ( $n, m$ )
  new (head)
  head  $\uparrow$ . no  $\leftarrow 1$ 
  p  $\leftarrow$  head
  for i  $\leftarrow 2$  to  $n$  do {初始化循环队列}
    new (q)
    q  $\uparrow$ . no  $\leftarrow i$ 
    q  $\uparrow$ . prev  $\leftarrow$  p
    p  $\uparrow$ . next  $\leftarrow$  q
    p  $\leftarrow$  q
  head  $\uparrow$ . prev  $\leftarrow$  p
  p  $\uparrow$ . next  $\leftarrow$  head
  p  $\leftarrow$  head
  for i  $\leftarrow 1$  to  $n - 1$  do
    for j  $\leftarrow 1$  to  $m$  do {报数}
      p  $\leftarrow$  p  $\uparrow$ . next
    H[i]  $\leftarrow$  p  $\uparrow$ . no {从队列中将报  $m$  的人删除}
    q  $\leftarrow$  p  $\uparrow$ . prev
    p  $\leftarrow$  p  $\uparrow$ . next
    q  $\uparrow$ . next  $\leftarrow$  p
    p  $\uparrow$ . prev  $\leftarrow$  q
  H[n]  $\leftarrow$  p  $\uparrow$ . no
  
```

## 2.5 小结

线性表及其链接表的基本算法和实现方法是基础,正是由于它的基础性,就更有牢固掌握它的必要。所谓“万丈高楼平地起”,线性表是所有算法的起始,是数据结构的最直接的呈现形态,很多数据结构都由线性表和链接表来实现。同样线性表和链接表也体现了物理存储的两种方式:连续线性和离散存储。正由于这些重要性,所以请读者务必熟练掌握。



## 习题二

### 一、单选题

1. 在一个长度为  $n$  的顺序存储的线性表中, 向第  $i$  个元素 ( $1 \leq i \leq n+1$ ) 位置插入一个新元素时, 需要从后向前依次后移( )个元素。  
A.  $n-i$                       B.  $n-i+1$                       C.  $n-i-1$                       D.  $i$
2. 在一个长度为  $n$  的顺序存储的线性表中, 删除第  $i$  元素 ( $1 \leq i \leq n$ ) 时, 需要从前向后依次前移( )个元素。  
A.  $n-i$                       B.  $n-i+1$                       C.  $n-i-1$                       D.  $i$
3. 在一个长度为  $n$  的线性表中顺序查找值为  $x$  的元素时, 在等概率情况下, 查找成功时的平均查找长度为( )。  
A.  $n$                       B.  $n/2$                       C.  $(n+1)/2$                       D.  $(n-1)/2$
4. 在一个长度为  $n$  的线性表中, 删除值为  $x$  的元素时需要比较元素和移动元素的总次数为( )。  
A.  $(n+1)/2$                       B.  $n/2$                       C.  $n$                       D.  $n+1$
5. 在一个顺序表的表尾插入一个元素的时间复杂度为( )。  
A.  $O(n)$                       B.  $O(1)$                       C.  $O(n^2)$                       D.  $O(\log_2 n)$
6. 在一个表头指针为  $ph$  的单链表中, 若要在指针  $q$  所指结点的后面插入一个由指针  $p$  所指向的结点, 则执行( )操作。  
A.  $q^{\wedge}.next = p^{\wedge}.next; p^{\wedge}.next = q;$                       B.  $p^{\wedge}.next = q^{\wedge}.next; q = p;$   
C.  $q^{\wedge}.next = p^{\wedge}.next; p^{\wedge}.next = q;$                       D.  $p^{\wedge}.next = q^{\wedge}.next; q^{\wedge}.next = p;$
7. 若某链表中最常用的操作是在最后一个结点之后插入一个结点和删除最后一个结点, 则采用( )存储方式最节省运算时间。  
A. 单链表                      B. 双链表  
C. 单循环链表                      D. 带头结点的双循环链表
8. 用单链表表示的链式队列的队头在链表的( )位置。  
A. 链头                      B. 链尾                      C. 链中                      D. 不确定

### 二、填空题

1. 对一个长度为  $n$  的线性表分别进行遍历和逆置运算, 其时间复杂度分别为\_\_\_\_\_和\_\_\_\_\_。
2. 若经常需要对线性表进行插入和删除运算, 则最好采用\_\_\_\_\_存储结构; 若经常需要对线性表进行查找运算, 则最好采用\_\_\_\_\_存储结构。
3. 由  $n$  个元素生成一个顺序表, 若每次都调用插入算法把一个元素插入到表头, 则整个算法的时间复杂度为\_\_\_\_\_; 若每次都调用插入算法把一个元素插入到表尾, 则整个算法的时间复杂度为\_\_\_\_\_。
4. 由  $n$  个元素生成一个单链表, 若每次都调用插入算法把一个元素插入到表头, 则整个算法的时间复杂度为\_\_\_\_\_; 若每次都调用插入算法把一个元素插入到表尾, 则

整个算法的时间复杂度为\_\_\_\_\_。

5. 对于一个长度为  $n$  的顺序存储的线性表, 在表头插入元素的时间复杂度为\_\_\_\_\_, 在表尾插入元素的时间复杂度为\_\_\_\_\_。

6. 对于一个长度为  $n$  的单链接存储的线性表, 在表头插入结点的时间复杂度为\_\_\_\_\_, 在表尾插入结点的时间复杂度为\_\_\_\_\_。

7. 在线性表的单链接存储中, 若一个元素所在结点的地址为  $p$ , 则其后继结点的地址为\_\_\_\_\_。

8. 访问一个长度为  $n$  的顺序表和单链表中第  $i$  个元素 (结点) 的时间复杂度分别为\_\_\_\_\_和\_\_\_\_\_。

### 三、上机编程题

#### 1. 津津的储蓄计划。

问题描述:

津津的零花钱一直都是自己管理。每个月的月初妈妈给津津 300 元钱, 津津会预算这个月的花销, 并且总能做到实际花销和预算的相同。

为了让津津学习如何储蓄, 妈妈提出, 津津可以随时把整百的钱存在她那里, 到了年末她会加上 20% 还给津津。因此津津制定了一个储蓄计划: 每个月的月初, 在得到妈妈给的零花钱后, 如果她预计到这个月的月末手中还会有多于 100 元或恰好 100 元, 她就会把整百的钱存在妈妈那里, 剩余的钱留在自己手中。

例如 11 月初津津手中还有 83 元, 妈妈给了津津 300 元。津津预计 11 月的花销是 180 元, 那么她就会在妈妈那里存 200 元, 自己留下 183 元。到了 11 月月末, 津津手中会剩下 3 元钱。

津津发现这个储蓄计划的主要风险是, 存在妈妈那里的钱在年末之前不能取出。有可能在某个月的月初, 津津手中的钱加上这个月妈妈给的钱, 不够这个月的原定预算。如果出现这种情况, 津津将不得不在这个月省吃俭用, 压缩预算。

现在请你根据 2004 年 1 月到 12 月每个月津津的预算, 判断会不会出现这种情况。如果不会, 计算到 2004 年年末, 妈妈将津津平常存的钱加上 20% 还给津津之后, 津津手中会有多少钱。

输入:

输入文件 save.in 包括 12 行数据, 每行包含一个小于 350 的非负整数, 分别表示 1 月到 12 月津津的预算。

输出:

输出文件 save.out 包括一行, 这一行只包含一个整数。如果储蓄计划实施过程中出现某个月钱不够用的情况, 输出  $-X$ ,  $X$  表示出现这种情况的第一个月; 否则输出到 2004 年年末津津手中会有多少钱。

样例输入 1:          样例输出 1

290                    -7

230

280

200

300

170

340

50

90

80

200

60

样例输入 2:            样例输出 2:

290

1580

230

280

200

300

170

330

50

90

80

200

60

2. 谁拿了最多奖学金?

问题描述:

某校的惯例是在每学期的期末考试之后发放奖学金。发放的奖学金共有五种，获取的条件各不相同：

1) 院士奖学金，每人 8000 元，期末平均成绩高于 80 分 ( $>80$ )，并且在本学期内发表 1 篇或 1 篇以上论文的学生均可获得；

2) 五四奖学金，每人 4000 元，期末平均成绩高于 85 分 ( $>85$ )，并且班级评议成绩高于 80 分 ( $>80$ ) 的学生均可获得；

3) 成绩优秀奖，每人 2000 元，期末平均成绩高于 90 分 ( $>90$ ) 的学生均可获得；

4) 西部奖学金，每人 1000 元，期末平均成绩高于 85 分 ( $>85$ ) 的西部省份学生均可获得；

5) 班级贡献奖，每人 850 元，班级评议成绩高于 80 分 ( $>80$ ) 的学生干部均可获得。

只要符合条件就可以得奖，每项奖学金的获奖人数没有限制，每名学生也可以同时获得多项奖学金。例如姚林的期末平均成绩是 87 分，班级评议成绩 82 分，同时他还是一位学生干部，那么他可以同时获得五四奖学金和班级贡献奖，奖金总数是 4850 元。

现在给出若干学生的相关数据，请计算哪些同学获得的奖金总数最高（假设总有同学能满足获得奖学金的条件）。

输入：

输入文件 scholar.in 的第一行是一个整数  $N$  ( $1 \leq N \leq 100$ )，表示学生的总数。接下来的  $N$  行每行是一位学生的数据，从左向右依次是姓名，期末平均成绩，班级评议成绩，是否是学生干部，是否是西部省份学生，以及发表的论文数。姓名是由大小写英文字母组成的长度不超过 20 的字符

串 (不含空格); 期末平均成绩和班级评议成绩都是 0~100 之间的整数 (包括 0 和 100); 是否是学生干部和是否是西部省份学生分别用一个字符表示, Y 表示是, N 表示不是; 发表的论文数是 0~10 的整数 (包括 0 和 10)。每两个相邻数据项之间用一个空格分隔。

输出:

输出文件 scholar.out 包括三行: 第一行是获得最多奖金的学生的姓名, 第二行是这名学生获得的奖金总数。如果有两位或两位以上的学生获得的奖金最多, 输出他们之中在输入文件中出现最早的学生的姓名。第三行是这 N 个学生获得的奖学金的总数。

样例输入:

4

YaoLin 87 82 Y N 0

ChenRuiyi 88 78 N Y 1

LiXin 92 88 N N 0

ZhangQin 83 87 Y N 1

样例输出:

ChenRuiyi

9000

28700

3. 断链取珠。

问题描述:

你有一条由 N 个红色的、白色的或蓝色的珠子组成的项链 ( $3 \leq N \leq 350$ ), 珠子是随意安排的。这里是  $n=29$  的两个例子:

12

```

      r b b r
      r      b
      r      r
      r      r
      b      r
      b      b
      b      b
      r      r
      b      r
      b      r
      r      r
      r b r
  
```

图 A

12

```

      b r r b
      b      b
      b      r
      w      r
      w      w
      r      r
      b      b
      r      b
      b      r
      r      r
      r      r
      r      b
      r r w
  
```

图 B

r 代表 红色的珠子 b 代表 蓝色的珠子 w 代表 白色的珠子

说明:

第一和第二个珠子在图片中已经被作记号。

图 A 中的项链可以用下面的字符串表示:

brbrrrbbrrrrrbrbrrbrrrrb.

请选择一些点剪断项链，展开成一条直线，然后从一端开始收集同颜色的珠子直到你遇到一个不同的颜色珠子，在另一端做同样的事（颜色可能与在这之前收集的不同）。确定应该在哪里剪断项链来收集到珠子的个数最多。

例：图 A 中的项链，可以收集到 8 个珠子，在珠子 9 和珠子 10 或珠子 24 和珠子 25 之间剪断项链。

注意：如果项链中包括有白色的珠子（如图 B），当收集珠子的时候，遇到的白色珠子可以被当做红色也可以被当做蓝色。表示项链的字符串将会包括三种符号 r、b 和 w。

请你写一个程序来确定从一条给定的项链最大可以被收集的珠子数目。

输入：

输入文件 wrb.in 有两行。第一行：N，为珠子的数目；第二行：一串长度为 N 的字符串。每个字符是 r、b 或 w，中间没有空格。

输出：

输出文件 wrb.out 只有一行，就是收集珠子数目的最大值。

样例输入：

29

wwwbbrwrbrbrrrbrrrrwrrbrrrrb

样例输出：

11

#### 4. 陶陶摘苹果。

问题描述：

陶陶家的院子里有一棵苹果树，每到秋天树上就会结出 10 个苹果。苹果成熟的时候，陶陶就会跑去摘苹果。陶陶有个 30 厘米高的板凳，当她不能直接用手摘到苹果的时候，就会踩到板凳上再试试。

现在已知 10 个苹果到地面的高度，以及陶陶把手伸直的时候能够达到的最大高度，请帮陶陶算一下她能够摘到的苹果的数目。假设她碰到苹果，苹果就会掉下来。

输入：

输入文件 apple.in 包括两行数据。第一行包含 10 个 100~200 之间（包括 100 和 200）的整数（以厘米为单位）分别表示 10 个苹果到地面的高度，两个相邻的整数之间用一个空格隔开。第二行只包括一个 100~120 之间（包含 100 和 120）的整数（以厘米为单位），表示陶陶把手伸直的时候能够达到的最大高度。

输出：

输出文件 apple.out 包括一行，这一行只有一个整数，表示陶陶能够摘到的苹果的数目。

样例输入：

100 200 150 140 129 134 167 198 200 111

110

样例输出：

5



## 3 栈和队列

### 3.1 栈

#### 一、栈的定义

栈 (stack) 又叫堆栈, 它是一种运算受限的线性表。其限制是仅允许在表的一端进行插入和删除运算。这一端被称为栈顶, 相对地, 把另一端称为栈底。向一个栈插入新元素又称作进栈、入栈或压栈, 它是把新元素放到栈顶元素的上面, 使之成为新的栈顶元素; 从一个栈删除元素又称作出栈或退栈, 它是把栈顶元素删除掉, 使其相邻的元素成为新的栈顶元素。

在日常生活中, 有许多类似栈的例子, 如刷洗盘子时, 把洗净的盘子一个接一个地向上放 (相当于进栈), 取用盘子时, 则从上面一个接一个地向下拿 (相当于出栈); 又如向自动步枪的弹夹装子弹时, 子弹被一个接一个地压入 (相当于进栈), 射击时总是后压入的先射出 (相当于出栈)。

由于栈的插入和删除运算仅在栈顶一端进行, 后进栈的元素必定先被删除, 所以又把栈称作后进先出表 (Last In First Out, 简称 LIFO)。

#### 二、栈的顺序存储

栈既然是一种线性表, 所以线性表的存储结构也同样适用于栈。栈的一种最简单的存储结构当然也是顺序存储。因此, 可把栈的顺序存储结构所使用的记录类型定义为:

```
type stack = record
    vec: array [1..m0] of elemtype;
    top: Integer
end
```

其中 vec 域用来顺序存储栈的元素, top 域用来存储栈顶元素所在单元的编号 (即下标), 所以又把 top 称为栈顶指针,  $m_0$  表示栈能够达到的最大深度 (即长度)。

设一个栈为  $T = (1, 2, 3, 4)$ , 栈 T 所对应的顺序存储结构如图 3-1 (a) 所示, 若在 T 中插入一个元素 5 或删除一个元素, 则分别对应的顺序存储结构如图 3-1 (b) 和 (c) 所示。在栈的顺序存储结构中, 为形象地使栈顶在上, 栈底在下, 所以采用的单元编号是向上递增的。

在一个栈中, 若 top 已经指向  $m_0$  单元, 则表示栈满; 若 top = 0, 则表示栈空。向一个满栈插入元素和从一个空栈删除元素都属于错误操作, 应当避免。

#### 三、栈的运算

栈的运算主要是插入和删除, 除此之外, 还有读取栈顶元素、置空栈和判断一个栈是否为空等。栈的运算都比较简单, 具体列出如下:

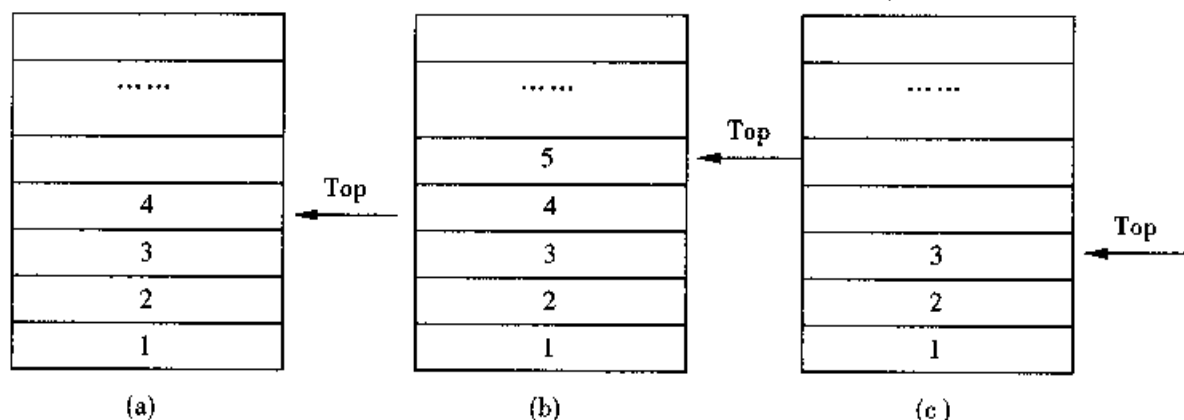


图 3-1 顺序栈的插入和删除示意图

- (1) 进栈 (push), 即向栈顶插入一个新元素;
- (2) 出栈 (pop), 即删除栈顶元素;
- (3) 读取栈顶元素 (readtop);
- (4) 置空栈 (setnull);
- (5) 判断一个栈是否为空 (empty)。

根据栈的运算和栈的顺序存储结构可写出相应的算法。设  $s$  为具有顺序存储结构的  $stack$  类型的一个栈,  $x$  为具有  $elemtype$  类型的一个数据元素, 则栈的各种运算所对应的算法如下:

#### 1. 进栈算法

算法步骤为:

- (1) 检查栈是否已满, 若栈满, 则进行“溢出”错误处理;
- (2) 将栈顶指针上移 (即加 1);
- (3) 将新元素赋给栈顶单元。

算法描述为:

push ( $S, x$ )

if  $S.top \leftarrow m0$  then error ( 'overflow' )

$S.top \leftarrow S.top + 1$

$S.vec [S.top] \leftarrow x$

此算法中的第 (1) 步不是必不可少的, 例如, 当你确信插入后不会造成溢出或者在调用此算法之前已经判断栈未满时, 则可省去此步。

#### 2. 出栈算法

算法步骤为:

- (1) 检查栈是否为空, 若栈空, 则进行“下溢”错误处理;
- (2) 将栈顶元素赋值给特定变参  $x$  (或者作为函数返回);
- (3) 将栈顶指针下移 (即减 1)。

算法描述为:

pop ( $S, x$ )

if  $S.top = 0$  then error ( 'underflow' )

$x \leftarrow S.vec [S.top]$

$S.top \leftarrow S.top - 1$

此算法的第(1)步有时也可以省略,如当你确信不会发生下溢,或者调用此算法之前已经判定栈未空时就是如此。从出栈算法可以看出:原栈顶元素仍然存在(即没有被破坏),只是栈顶指针不指向它,而指向了它下面的元素。

### 3. 读取栈顶元素的算法

此算法很简单,若不考虑栈空的情况,只要将栈顶元素赋给指定变参或函数名即可。假定写成函数的形式,则为:

```
readtop (S)
    return (S.vec [S.top])
```

注意:在这个算法中栈顶指针保持不变。

### 4. 置空栈算法

此算法也很简单,只要把栈顶指针置0即可。算法描述为:

```
setnull (S);
    S.top ← 0
```

### 5. 判断一个栈是否为空栈的算法

此算法可用一个函数来描述,栈空时返回“真”值,非空时返回“假”值。

```
empty (S): boolean
    if S.top = 0
        then empty ← true
        else empty ← false
```

在上面栈的各种算法中,都不需要进行元素的比较和移动,所以其时间复杂性均为  $O(1)$ 。

## 四、双栈操作

当在一个程序中需要同时使用具有相同成分类型的两个栈时,一种最直接最简单的方法是为每个栈分别开辟一个存储空间,不过这样做的结果可能出现一个栈的空间已被占满而无法再进行插入操作,另一个栈的空间仍有大量剩余而没有得到利用的情况,从而造成存储空间的浪费;另一种可取的方法是为两个栈共同开辟一个存储空间,让一个栈的栈底为该空间的始端,另一个栈的栈底为该空间的末端,当元素进栈时,都从两端向中间“增长”,这样能够使剩余的空间为任一栈所使用,即当一个栈的深度不足整个空间的一半时,另一个栈的深度可超过其一半,从而提高了存储空间的利用率。

两个栈共用一个存储空间的顺序存储结构所使用的记录类型可定义为:

```
type bothstack = record
    vec : array [1..m0] of elementype;
    topl, top2 : integer
end;
```

其中 topl 和 top2 分别为栈1和栈2的栈顶指针,  $m_0$  为整个存储空间的大小(即所含的存储单元数)。

设有两个栈:

$T_1 = (a_1, a_2, \dots, a_n)$

$T_2 = (b_1, b_2, \dots, b_m)$

若它们共同使用具有 bothstack 类型的一个双栈空间时, 其对应的顺序存储结构如图 3-2 所示。

在这种双栈操作中, 当对任一栈进行插入时, 若  $n + m = m$  (即  $top1 = top2 - 1$  或  $top2 = top1 + 1$ ), 则表示栈满; 当对栈 1 或栈 2 进行删除时, 若  $top1 = 0$  或  $top2 = m_0 + 1$ , 则表明相应的栈空。另外, 当新元素压入栈 2 时, 栈顶指针  $top2$  不是增 1 而是减 1; 当从栈 2 删除元素时,  $top2$  不是减 1 而是增 1。

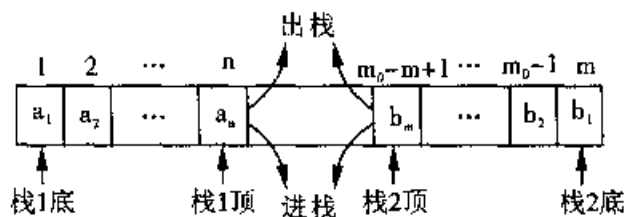


图 3-2 双栈结构示意图

设 BS 表示具有 bothstack 类型的双栈,  $x$  表示具有 elemtype 类型的参数,  $k$  表示整型参数, 它只取 1 和 2 两个值。当  $k=1$  时, 表示对栈 1 操作; 当  $k=2$  时, 表示对栈 2 操作。下面分别给出对任一栈进行插入、删除和置空栈的算法。

#### 1. 插入算法

push (BS,  $k$ ,  $x$ )

if BS.top1 = BS.top2 - 1 then error ( 'overflow' )

if  $k = 1$

then

BS.top1 ← BS.top1 + 1

BS.vec [ BS.top1 ] ←  $x$

else

BS.top2 ← BS.top2 - 1

BS.vec [ BS.top2 ] ←  $x$

#### 2. 删除算法

下面以函数的形式给出

pop (BS,  $k$ )

if  $k = 1$

then (对第一个栈进行删除)

if BS.top1 = 0 then error ( 'underflow' )

pop ← BS.vec [ BS.top1 ]

BS.top1 ← BS.top1 - 1

else {对第二个栈进行删除}

if BS.top2 = m0 + 1 then error ( 'underflow' )

pop ← BS.vec [ BS.top2 ];

BS.top2 ← BS.top2 + 1

## 3. 置空栈算法

```
setnull (BS, k)
```

```
  if k = 1
```

```
    then BS.top ← 0
```

```
    else BS.top2 ← m0 + 1
```

当利用一个存储空间定义三个或三个以上的栈时，只有迎面增长的栈之间才能够互补余缺，而背向增长或同向增长的栈之间就无法自动地互补余缺了。因此要充分地利利用剩余的空间，就必须对某些栈作整体移动，这将是浪费时间的，是不可取的。在后面几节我们将会看到，采用链接堆栈能够有效地克服这种缺点。

## 3. 2 栈的应用举例

栈在计算机科学领域具有广泛的应用。比如，在编译和运行计算机语言程序的过程中，就需要利用栈进行语法检查（如检查 begin 和 end，“（”和”）、“[”和”]”是否配对等）、计算表达式的值、实现递归过程和函数的调用等。下面举例说明栈在这些方面的应用。

**例题 3-1** 设一个字符型数组 B 中存放着以特定字符 @ 作为结束符的表达式，试编写一个检查表达式中左、右圆括号是否配对的函数算法。若配对，则返回“真”值，否则返回“假”值。

**分析** 此算法可使用一个栈，假定用 S 表示，类型为 stack，成分类型为字符，用它来存储表达式中从左到右顺序扫描得到的左括号，栈的最大深度不会超过表达式中所含的左括号的个数。此算法的基本思路是：顺序扫描数组 B 中的每一个字符，若遇到的是左括号 ‘（’，则令其进栈，若遇到的是右括号，则令其相配的左括号（即栈顶元素）出栈，当 S 栈发生下溢或当表达式处理完毕而 S 栈非空时，则表明括号不配对，应返回“假”值，否则返回“真”值。

此算法描述如下：

```
Function Pair (B): Boolean;
```

```
begin
```

```
  SetNull (S)
```

```
  i ← 1
```

```
  ch ← B [i]
```

```
  while ch ≠ '@' do
```

```
    if (ch = '（') or (ch = '）') then
```

```
      case ch of
```

```
        '（': Push (S, ch)
```

```
        '）': if empty (S) then return (false)
```

```
            else Pop (S)
```

```
      end case
```

```
    i ← i + 1
```

```
    ch ← B [i]
```

```
  if empty (S) then return (true)
```

else return (false)

end;

例题 3-2 表达式的计算。

### 1. 算术表达式的两种表示

通常书写的算术表达式是把运算符放在两个操作数（又叫运算对象或运算量）的中间。如对于  $2 + 5 * 6$ ，乘法运算符“ $*$ ”，的两个操作数是它两边的 5 和 6；加法运算符“ $+$ ”的两个操作数，一个是它前面的 2，另一个是它后面的  $5 * 6$  即 30。我们把算术表达式的这种表示叫做中缀表示，采用中缀表示的算术表达式简称中缀算术表达式。

中缀表达式的计算必须按照以下三条规则进行：

(1) 先算括号内，后算括号外；

(2) 在无括号或同层括号内的情况下，先做乘除运算，后做加减运算，即乘除运算的优先级高于加减运算的优先级（假定我们只讨论这四种运算）；

(3) 同一优先级运算，从左向右依次进行。

从这三条规则可以看出，在中缀表达式的计算过程中，既要考虑括号的作用，又要考虑运算符的优先级，还要考虑运算符出现的先后次序。因此，各运算符实际的运算次序往往同它们出现的先后次序是不一致的，是不可预测的。当然凭肉眼判别一个中缀表达式中哪个运算符最先算，哪个次之……哪个最后算并不困难，但通过计算机处理就比较麻烦了，因为计算机只能一个字符一个字符地扫描，要想得到哪一个运算符先算，就必须对整个中缀表达式扫描一遍，一个中缀表达式中有多少个运算符，原则上就得扫描多少遍才能计算完毕，这样就太浪费时间了，显然是不可取的。

那么，能否把中缀算术表达式转换成另一种形式的算术表达式，使计算简单化呢？

回答是肯定的。波兰科学家卢卡谢维奇（Lukasiewicz）很早就提出了算术表达式的另一种表示，即后缀表示，又称逆波兰式，其定义是把运算符放在两个运算对象的后面。采用后缀表示的算术表达式简称后缀算术表达式或后缀表达式。在后缀表达式中，不存在括号，也不存在优先级的差别，计算过程完全按照运算符出现的先后次序进行，整个计算过程仅需一遍扫描便可完成，显然比中缀表达式的计算要简单得多。例如，对于后缀表达式  $12\_4\_ - 5\_ /$ ，其中“ $\_$ ”表示空格字符，因减法运算符在前，除法运算符在后，所以应先做减法，后做除法；减法的两个操作数是它前面的 12 和 4，其中第一个数 12 是被减数，第二个数 4 是减数；除法的两个操作数是它前面的 12 减 4 的差（即 8）和 5，其中 8 是被除数，5 是除数。

中缀算术表达式转换成对应的后缀算术表达式的规则是：把每个运算符都移到它的两个运算对象的后面，然后删除掉所有的括号即可。

例如，对于下列各中缀表达式：

(1)  $3/5 + 6$

(2)  $16 - 9 * (4 + 3)$

(3)  $2 * (x + y) / (1 - x)$

(4)  $(25 + x) * (a * (a + b) + b)$

对应的后缀表达式分别为：

(1)  $3\_5\_ / 6\_ +$

(2)  $16\_9\_4\_3\_ + * -$



(3)  $2\_x\_y + *1\_x\_ - /$

(4)  $25\_x\_ + a\_ a\_ b\_ + *b\_ + *$

下面我们将讨论后缀表达式求值的算法。为了讨论方便,更好地突出问题的本质,不妨假定算术表达式中的每个操作数都是大于等于 0 的整数,并且算术表达式的语法都是正确的,因而不需要在算法中进行语法检查。

## 2. 后缀算术表达式求值的算法

**分析** 设定一个栈 S1, 类型为 stack, 成分类型为实型 (real), 用它来存储运算的操作数、中间结果以及最后结果。作为此算法的输入, 假定在一个字符型数组 A 中已存放着一个以 @ 字符作为结束符的后缀算术表达式, 并且该表达式中的每个操作数都是以空格字符结束的。此算法的基本思路是: 从数组 A 中的第一个字符起扫描, 若遇到的是数字字符, 则就把以它开头的一组数字字符 (直到遇到空格字符为止) 转换成对应的数值 (即一个操作数) 后压入 S1 栈; 若遇到的是运算符, 则就从 S1 栈顶依次弹出两个操作数, 进行相应的运算后, 再把结果压入 S1 栈; 继续扫描下一个字符并处理, 直到遇到 @ 字符为止。算法结束后, S1 栈顶的值就是数组 A 中后缀算术表达式的计算结果。假定该结果由函数名带回, 则算法描述为:

```
Function Comp (A): real;
Begin
    Setnull (S1)
    i ← 1
    ch ← A [i]
    while ch < > '@' do
        case ch of
            '0' .. '9': x ← 0
                while ch < > ' ' do
                    x ← x * 10 + ord (ch) - ord ('0')
                    i ← i + 1
                    ch ← A [i]
            '+': x ← pop (S1) + pop (S1)
            '-': x ← pop (S1)
                x ← pop (S1) - x;
            '*': x ← pop (S1) * pop (S1)
            '/': x ← pop (S1)
                x ← pop (S1) / x
        End case
        Push (S1, x)
        i ← i + 1
        ch ← A [i]
    return (pop (S1))
end;
```

此算法的运行时间主要花在扫描上, 算法从头到尾扫描并处理数组 A 中字符, 若后缀算术表

达式由  $n$  个字符组成, 则此算法的时间复杂性为  $O(n)$ 。此算法在运行时所占用的临时空间主要取决于 SI 栈的大小, 显然, 它的最大深度不会超过表达式操作数的个数。若操作数的个数为  $m$ , 则此算法的空间复杂性为  $O(m)$ 。

在这个算法中, 若数组 A 的后缀表达式为:

14\_ 3\_ 20\_ 5/\_ \* 8\_ - + @

则从第四个操作数开始, 每处理一个操作数或运算符后, SI 栈的状态如图 3-3 所示。

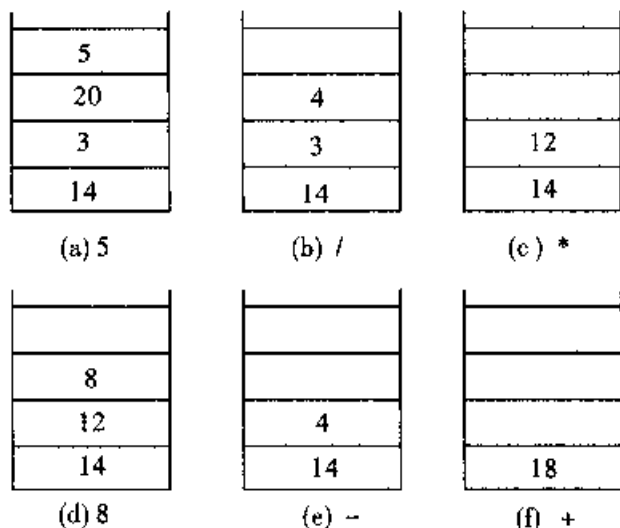


图 3-3 SI 栈的数据变化

### 3. 把中缀表达式转换成后缀表达式的算法

中缀算术表达式转换成对应的后缀算术表达式的规则是: 把每个运算符都移到它的两个运算对象的后面, 然后删除掉所有的括号即可。

将下列的中缀表达式改写成后缀表达式:

- (1)  $3/5+6$                       (2)  $16-9*(4+3)$   
 (3)  $2*(x+y)/(1-x)$         (4)  $(25+x)*(a*(a+b)+b)$

结果为:

- (1) 3\_ 5\_ /6\_ +                      (2) 16\_ 9\_ 4\_ 3+ \* -  
 (3) 2\_ x\_ y+ \* 1\_ x\_ - /        (4) 25\_ x+a\_ a\_ b\_ + \* b\_ + \*

分析两种表达式的异同: 操作数出现的先后顺序没有变, 只有运算符变了, 怎么变的呢? 还是根据中缀表达式的三条运算规则, 有多少运算符就扫描多少遍就能确定每个运算符出现的先后顺序, 从而得到后缀表达式。同样, 能否优化这个算法, 减少扫描的次数, 更快捷得到后缀表达式呢?

回答是肯定的, 这要从中缀表达式运算符的优先级入手。

在中缀算术表达式中, 设  $P_1$  和  $P_2$  分别表示前后相邻出现的两个算符, 那么  $P_1$  和  $P_2$  之间的优先关系只能是下列两种情况之一:

$P_1 < P_2$  表示  $P_1$  落后于  $P_2$  运算或者说  $P_2$  优先于  $P_1$  运算

$P_1 > P_2$  表示  $P_1$  优先于  $P_2$  运算或者说  $P_2$  落后于  $P_1$  运算

根据中缀算术表达式运算的三条规则, 相邻算符之间的优先关系如表 3-1 所示。



表 3-1 相邻算符优先关系表

P1 \ P2	+	-	*	/	(	)	@
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
@	<	<	<	<	<		=

分析：设以@字符结束的一个中缀算术表达式已存放在字符型数组 E 中，而转换后生成的后缀算术表达式用字符数组 A 来存放。在转换过程中需要使用一个栈，假定用 S2 表示，用它来暂存不能立即送入数组 A 中的算符（这里暂且把运算符、左右圆括号和表达式结束符统称为算符），因此，S2 栈的成分类型应为字符型，S2 栈的最大深度不会超过中缀算术表达式中算符的个数。

当把中缀表达式转换成后缀表达式时，为了栈处理的方便，往往在栈底首先放入一个表达式结束符@，并令它具有最低的优先级（即落后于其他任何算符），当栈处理结束时，会出现表达式最后的@字符同栈底的@字符相遇的情况，此时表明中缀表达式已经转换完毕，所以在表中把它们定义为“=”关系。在表达式处理中，有时还可能遇到右括号与栈顶的左括号比较优先关系的情况，所以必须对它们给予定义，由于它们在算术表达式中只能同时存在和消失，所以在表中也把它们定义为“=”关系。在表中还有三处空白，这是 P1 和 P2 算符不应该相遇的情况，若发生这种情况，则表明中缀算术表达式中有语法错误。

把中缀表达式转换成对应的后缀表达式的基本思路是：从数组 E 的第一个字符起扫描，当遇到的是数字字符时，就把以它开始的一组数字字符（直到非数字字符为止）和附加的一个空格字符依次送入数组 A；当遇到的是算符时，首先检查它是否落后于栈顶算符的运算，若是，则表明栈顶算符的两个运算对象已经被送入到数组 A 中，应将栈顶算符退栈并送入数组 A，以此反复进行，直到不落后于栈顶算符为止，接着检查它是否优先于栈顶算符，若是，则表明它的第二个运算对象还没有被送入数组 A 中，所以应把它进栈，否则表明该算符必然是右括号，栈顶算符必然是左括号，因该括号内的算符已处理完毕，所以应把左括号从栈顶退出；继续扫描和处理下一个字符，直到遇到结束符@后，再做必要的结束处理即可。

算法描述为：

procedure change (E, A)

Begin

- (1) setnull (S2); push (S2, '@');
- i: = 1; {i 作为扫描数组 E 的指针}
- j: = 1; {j 用来指示数组 A 中待写入字符的位置}
- ch: = E[i]; {E 中第一个字符送给 ch}
- (2) while ch <> '@' do
- (I) if ch in op1 then {op1 为数字字符的集合}

```

(a) while ch in op1 do
    [ A [j]: = ch; j: = j+1; i: = i+1; ch: = E [i];
(b) A [j]: = ' '; j: = j+1;
    { 给 A 中的每个操作数后附加一个空格 }
(II) if ch in op2 then { op2 为算符的集合 }
    (a) w: = readtop (S2);
        while precede (w, ch) = '>' do
            { precede 为比较两算符优先关系的函数 }
            [ A [j]: = w; j: = j+1; pop (S2); w: = readtop (S2) ];
    (b) if precede (w, ch) = '<' then push (S2, ch)
        else pop (S2);
(III) i: = i+1; ch: = E [i];
(3) (I) w: = pop (S2);
    While w <> '@' do
    [ A [j]: = w; j: = j+1; w: = pop (S2) ];
    (II) A [j]: = '@'

```

End;

此算法的运行时间主要花在第(2)步上,如果单从这一步的 while 外循环来看,其时间复杂性为  $O(n)$ 。那么循环体内的每一个 while 循环和 precede 函数是否会增加此算法的时间复杂性呢?从第(I)步看,显然是不会的,因为每一次循环都将顺序处理数组 E 中的下一个字符;从第(II)步看,也是不会的,因为这里调用的 precede 函数实际上就是根据两个算符到表 3-1 中去查找其对应的优先关系,这种查找所需时间是固定的,它不随处理问题的规模(在此是指中缀算术表达式中包含字符的多少)而改变,在第(III)步中出现的 while 循环在任何情况下都不会超过两次循环。因此,整个算法的时间复杂性为  $O(n)$ 。

在这个算法中,若数组 E 中的中缀表达式为:

$2 * ((8 + 7) / 15) + 9 @$

则每处理一个算符后,数组 A 和栈 S2 的状态如图 3-4 所示。

从上面算术表达式的计算过程可以看出,利用后缀表示和栈技术只需两遍扫描即可完成,其中第一遍是把算术表达式的中缀表示转换成对应的后缀表示,第二遍是根据后缀表示进行求值。显然它比直接利用中缀算术表达式进行计算的扫描次数要少得多,算法的复杂性和编程的复杂性大大降低。

**例题 3-3** 编写一个算法,对于给定的十进制正整数,打印出相应的八进制整数。

**分析** 把十进制正整数转换成对应的八进制整数采用逐次除 8 取余法,即用基数 8 不断地去除被转换的十进制正整数,直到商为 0。设转换后得到的八进制整数为  $k_m k_{m-1} \cdots k_1 k_0$  ( $m \geq 0$ ),则第一次相除所得余数就是八进制整数的最低位  $k_0$ ,第二次相除所得余数就是八进制整数的次最低位  $k_1$ ,依此类推,最后一次相除所得余数(即商为 0 时的余数)就是八进制整数的最高位  $k_m$ 。

例如,把十进制正整数 765 转换成对应的八进制整数的过程如图 3-5 所示。

设被转换的十进制正整数用  $n$  表示,每次整除 8 所得余数用  $k$  表示,若采用递归的方法来编写此题的算法,则具体步骤为:

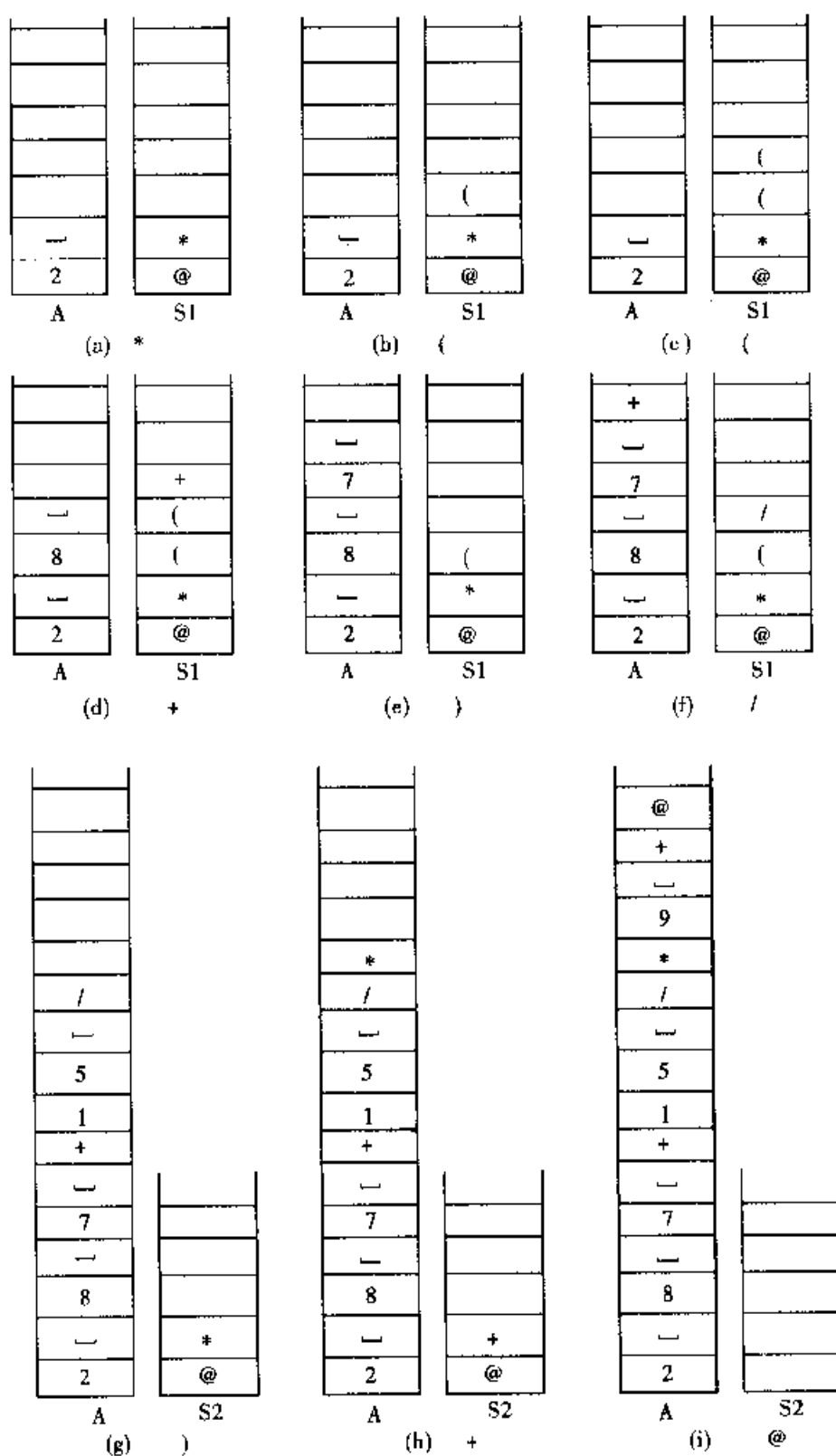


图 3-4 数组 A 和栈 S2 的数据变化

8	765	余数	八进制位
8	95	..... 5	$k_0$
8	11	..... 7	$k_1$
8	1	..... 3	$k_2$
	0	..... 1	$k_3$

图 3-5 十进制数 765 转换成八进制数的过程

- (1) 用 8 对  $n$  作取余运算, 将结果赋给  $k$ ;
- (2) 用 8 去整除  $n$ , 将结果仍赋给  $n$ ;
- (3) 如果  $n$  不为 0, 则以  $n$  为实参调用本过程;
- (4) 按域宽为 1 打印  $k$  的值。

用伪代码编写出此递归过程的算法为:

```

transfer (n)
k ← n mod 8
n ← n div 8
if n < > 0 then transfer (n)
write (k; 1)

```

计算机执行递归算法 (包括递归过程和递归函数) 时, 是通过栈来实现的。具体地说, 在运行开始时, 首先为递归调用建立一个栈, 该栈的成分类型 (即元素类型) 包括值参域、局部变量域和返回地址域; 在每次执行递归调用语句或函数之前, 自动把本算法中所使用的值参和局部变量的当前值以及调用后的返回地址压入栈; 在每次递归调用语句或函数执行结束后, 又自动把栈顶各域的值分别赋给相应的值参和局部变量, 以便使它们恢复为调用前的值, 接着无条件转向 (即返回) 由返回地址域所指定的位置执行。

对于调用上面的递归过程来说, 系统首先为后面的递归调用建立其成分类型包含值参  $n$  的域、局部变量  $k$  的域和返回地址的域的一个栈; 在每次执行 `transfer (n)` 语句递归调用前, 自动把  $n$  和  $k$  的当前值以及 `write` 语句的开始位置 (即调用后的返回地址) 压入栈; 在每次执行到最后的 `end` (即一次递归调用结束) 后, 又自动把栈顶的与  $n$  和  $k$  对应域的值分别赋给  $n$  和  $k$ , 接着无条件转向 `write` 语句的开始位置, 继续向下执行。

上面的分析说明了系统是如何利用堆栈技术来进行递归处理的。下面我们将讨论如何模拟系统处理递归的方法把一个递归算法改写成一个非递归的算法, 从而进一步加深对堆栈和递归这两个重要概念的理解和认识。

设  $p$  是一个递归算法, 假定  $p$  中共有  $m$  个值参数和局部变量, 共有  $t$  处递归调用  $p$  的过程语句或函数引用, 则把  $p$  改写成一个非递归算法的一般规则为:

- (1) 定义一个栈  $s$ , 用来保存每次递归调用前值参和局部变量的当前值以及调用后的返回地址,  $s$  栈中的成分类型应包含  $m+1$  个域, 其中前  $m$  个域为值参和局部变量而设, 后一个域为返回地址而设,  $s$  栈的深度应足够大, 使得在递归调用中不会发生溢出;
- (2) 定义  $t+2$  个语句标号, 其中用一个标号标在原算法中的第一条语句上, 用另一个标号标在作返回处理的第一条语句上, 其余  $t$  个标号作为  $t$  处递归调用的返回地址, 分别标在相应的语句上;

(3) 把每一个递归调用的语句或函数改写为:

(I) 把值参和局部变量的当前值以及调用后的返回地址压入栈;

(II) 把值参所对应的实在参数表达式的值赋给值参变量;

(III) 无条件转向原算法的第一条语句;

(4) 在算法结尾之前增加返回处理, 当栈非空时做:

(I) 退栈;

(II) 把原栈顶中前  $m$  个域的值分别赋给各对应的值参和局部变量;

(III) 无条件转向由本次返回地址所指定的位置;

(5) 增设一个同  $s$  栈的成分类型相同的变量, 作为进出栈的缓冲变量, 对于递归函数, 还需要再增设一个保存函数值中间结果的临时变量, 用这个变量替换函数体中所有函数名, 待函数过程结束之前, 再把这个变量的值赋给函数名返回;

(6) 在原算法的第一条语句之前, 增加一条置栈空的语句;

(7) 对于递归函数而言, 若某条赋值语句中包含有两处或多处 (假定为  $n$  处,  $n \geq 2$ ) 递归调用, 则应首先把它拆成  $n$  条赋值语句, 使得每条赋值语句中只包含一处递归调用, 同时对增加的  $n-1$  条赋值语句, 要增设  $n-1$  个局部变量, 然后再按照以上六条规转换成非递归函数。

### 3.3 队 列

#### 一、队列的定义

队列 (queue) 简称队, 它也是一种运算受限的线性表, 其限制是仅允许在表的一端进行插入, 而在表的另一端进行删除。我们把进行插入的一端称作队尾 (rear), 进行删除的一端称作队首 (front)。向队列中插入新元素称作进队或入队, 新元素进队后就成为新的队尾元素; 从队列中删除元素称作出队, 出队后, 其后继元素成为队首元素。由于队列的插入和删除分别在表的两端进行, 所以要删除的元素是队列中最先进入的元素, 因此又把队列称作先进先出 (First In First Out, 简称 FIFO) 表。

在日常生活中, 人们为购物或等车时所排的队就是一个队列, 新来购物或等车的人接到队尾 (即进队), 站在队首的人购物或上车后离开 (即出队), 当最后一人购物或上车离队后, 则队列为空。

#### 二、队列的顺序存储

队列的一种最简单的存储结构当然也是顺序存储, 所使用的记录类型可定义为:

```
type queue = record
    vec: array [1..m0] of elemtype;
    f, r: integer
end;
```

其中  $vec$  域用来存储队列的元素,  $f$  和  $r$  域分别用来存储队首元素和队尾元素所在单元的编号, 因此又把  $f$  和  $r$  分别称作队首指针和队尾指针。图 3-6 给出了一个队列在顺序存储方式下的当前状态, 此时已有  $a$ 、 $b$ 、 $c$  三个元素相继出队 (为了同队列中的元素相区别, 把它们分别括了起

来), 队首指针  $f$  指向队首元素  $d$ , 队尾指针  $r$  指向队尾元素  $j$ ; 若在图 (a) 的队中插入一个新元素  $k$ , 或者删除一个元素后, 队列的当前状态分别如图 (b) 和 (c) 所示。

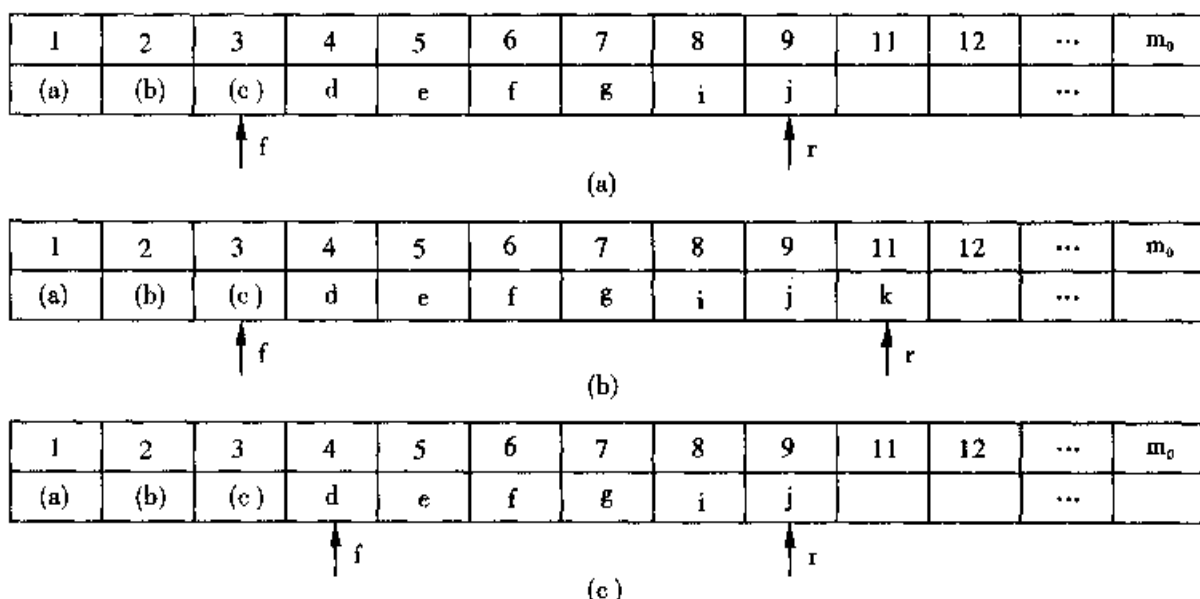


图 3-6 顺序队列的插入和删除示意图

每次向队列插入一个元素, 都将使队尾指针后移一个位置, 当队尾指针指向最后一个位置  $m_0$  时, 表明队列已满 (实际上, 若  $f$  不等于 1 的话, 队首前面仍有空闲的单元可再被利用), 若再进行插入, 则溢出 (我们把这种溢出叫做“假溢出”); 每次删除队列中的一个元素, 也同样使队首指针向后移动一个位置, 当队首指针指向最后一个元素 (即队尾指针所指的元素) 并被删除后, 表明队列已空, 此时应把  $f$  和  $r$  同时置为 0, 以便从第一个单元起重新利用整个存储空间。

### 三、队列的运算

队列的主要运算是插入、删除、置空队等。设  $Q$  为一个顺序存储的队列, 其类型为 `queue`,  $x$  为具有 `elemtype` 类型的一个参数, 则队列的插入、删除和置空队的算法分别如下:

#### 1. 插入算法

`insert (Q, x)`

```

if Q.r = m0 then error ( 'overflow' )
Q.r ← Q.r + 1 { 队尾指针后移 }
Q.vec [ Q.r ] ← x { 新元素赋给队尾单元 }
if Q.f = 0 then Q.f ← 1 { 若原为空队, 则进行插入后, 同时把队首指针置为 1 }
    
```

#### 2. 删除算法

此算法既可写成过程的形式, 也可写成函数的形式, 若写成过程的形式为:

`delete (Q, x)`

```

if Q.f = 0 then error ( 'underflow' )
x ← Q.vec [ Q.f ] { 把队首元素赋给 x }
if Q.f = Q.r
then
    Q.f ← 0
    
```

```

    Q.r ← 0
else
    Q.f ← Q.f + 1

```

### 3. 置空队算法

此算法很简单，只要把队首和队尾指针均赋 0 即可。

```
setnull (Q)
```

```
Q.f ← 0
```

```
Q.r ← 0
```

为了简化队列的插入和删除操作，我们假定队首指针始终指向队首的前一个位置，如图 3-7 所示。

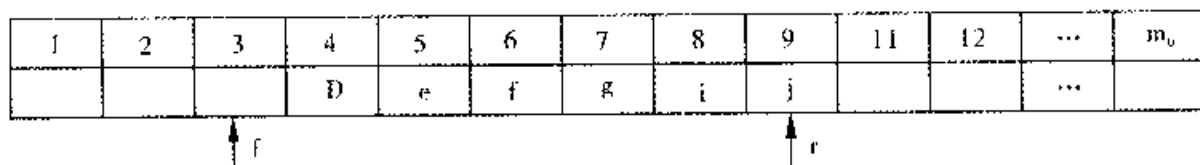


图 3-7 队首指针指向队首元素前一位置的顺序队列

这样，判断队空的条件应由  $Q.f = 0$  改为  $Q.f = Q.r$ ；进行删除时，要先移动队首指针，后取队首元素。进行插入时，若原队为空，则插入后队首指针正好指向队首元素的前一个位置，因此可省去上面插入算法中的第 (4) 步。改进后的插入和删除算法如下：

```
insert (Q, x)
```

```
if Q.r =  $m_0$  then error ( 'overflow' )
```

```
Q.r ← Q.r + 1
```

```
Q.vec [ Q.r ] ← x
```

```
delete (Q, X)
```

```
if Q.f = Q.r then error ( 'underflow' )
```

```
Q.f ← Q.f + 1
```

```
X ← Q.vec [ Q.f ]
```

改进后的算法虽然简单了，但“假溢出”的问题仍没有得到解决，因此还需要进一步改进。一种改进的方法是：当出现“假溢出”（即队尾达到最后  $m_0$  单元，而队首前面仍有空闲单元）时，把整个队列前移至始端，把所有空闲的单元留在后面，以便进行继续插入，这种方法将花费较多的时间用于数据移动。第二种改进的方法是采用循环队列，即把存储队列的整个数组空间看作是首尾相接的一个环，当队尾指针指向  $m_0$  单元后再进行插入时，把新元素插入到第一个单元中（若该单元空闲的话）。这种方法不需要移动任何元素，显然比第一种方法要好。循环队列也很容易实现，只要把队尾指针对  $m_0$  取模后加 1 即可。在循环队列中进行插入时，应把原来判断溢出的条件改为  $(Q.r \bmod m_0) + 1 = Q.f$ ，此时虽然队首指针  $f$  所指的单元空闲着，但若利用了它，就无法用  $Q.f = Q.r$  作为判断队空的条件了（因为可能是队满的情况）。所以，在循环队列中，实际可用的存储单元数为  $(m_0 - 1)$  个。

循环队列的插入和删除算法如下：

```
insert (Q, x)
```

```

if (Q.r mod m0) + 1 = Q.f then error ( 'overflow' )
Q.r ← (Q.r mod m0) + 1
    {队尾指针后移, 第  $m_0$  单元的后面是第 1 单元}
Q.vec [Q.r] ← x
delete (Q, x)
if Q.f = Q.r then error ( 'underflow' )
Q.f ← (Q.f mod m0) + 1
x ← Q.vec [Q.f]

```

顺便说一下, 循环队列的置空队算法和非循环队列相同, 都是把队首和队尾指针置为 0。

### 3.4 队列的应用举例

#### 例题 3-4 合并石子。

小 Ray 在河边玩耍, 无意中发现一些很漂亮的石子堆, 于是他决定把这些石子搬回家。河滩上共有  $n$  ( $1 \leq n \leq 30000$ ) 堆石子, 每次小 Ray 合并两个石子数最少的两堆石子成为一堆。经过  $n-1$  次合并操作以后, 只剩下一堆石子, 然后小 Ray 就将这一堆石子搬回家。每合并两堆石子的时候, 小 Ray 消耗的体力是两堆石子的数量之和。请你算一算, 小 Ray 合并所有石子堆消耗的体力是多少呢?

解析:

设这些石子堆的数量为  $w_1, w_2, \dots, w_n$ , 且满足  $w_1 \leq w_2 \leq \dots \leq w_n$ 。首先小 Ray 肯定将 1 和 2 两堆石子合并, 设新合并的石子堆为  $y_1 = w_1 + w_2$ 。接着小 Ray 一定是在  $\{y_1\} \cup \{w_2 \dots w_n\}$  中选择两个最小的石子堆合并, 那么设新合并的石子堆为  $y_2$ 。如此类推, 第三次合并的石子堆记作  $y_3$ , 第四次合并的记作  $y_4$ ... 第  $n-1$  次合并的石子堆记作  $y_{n-1}$ 。

可以证明, 新合并的石子堆的大小一定满足:  $y_1 \leq y_2 \leq y_3 \leq \dots \leq y_{n-1}$ 。因为每次合并的新石子堆一定是选择当前所剩下的石子堆中最小的两堆合并, 而剩下的石子堆的石子数量又是不断增多的, 所以越早合并的石子堆的石子数量越少。

有了上面这条事实, 我们知道在合并过程中  $w$  和  $y$  数组始终是保持有序的。因此假设当前剩下的石子堆为  $w_1 \dots w_n$  和  $y_1 \dots y_m$ , 那么最小的石子堆不是  $w_1$  就是  $y_1$ , 拿出其中最小的一个, 然后再拿出次小的一个, 合并成新的石子堆一定是放在  $y_m$  之后。也就是说将  $y$  看成一个队列, 在队首取出元素, 在队尾插入元素, 且在插入和删除的过程队列始终保证了从小到大的顺序 (越靠近队首越小)。由于  $w$  中的每个元素最多删除一次,  $y$  中的元素最多插入队列和从队列中取出一次, 所以总的复杂度为  $O(n)$ 。

#### 例题 3-5 马的遍历。

在一个  $n \times n$  的棋盘上有一匹马站在第  $x$  行第  $y$  列的格子上。棋盘上有些格子有障碍物, 用 “\*” 表示, 马不能够站在有障碍物的格子上。问: 马按照国际象棋中的走法能够到达棋盘上的哪些格子, 且到达这些格子最少的步数是多少?

马在国际象棋中的走法如下: 如果马站在  $(x, y)$  上, 那么马可以走到的格子是  $(x+2, y+1)$





1),  $(x+2, y-1)$ ,  $(x+1, y+2)$ ,  $(x+1, y-2)$ ,  $(x-1, y+2)$ ,  $(x-1, y-2)$ ,  $(x-2, y+1)$ ,  $(x-2, y-1)$ 。具体如图 3-8 所示:

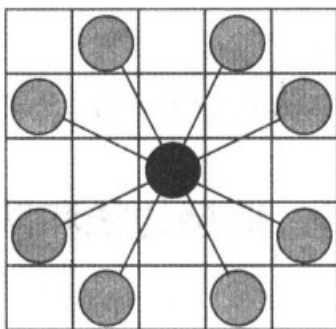


图 3-8 跳马示意图

输入: 第一行为  $n, x, y$ , 接下来  $n$  行为棋盘的描述。“\_”表示空格子, “\*”表示有障碍物。

输出: 一共  $n$  行, 每行  $n$  个数, 表示马到这个格子最少需要的步数, 如果无法到达用  $-1$  表示。

解析:

这种遍历很容易用队列实现。首先队列中只有  $(x, y)$  这个点, 然后从  $(x, y)$  这个点扩展, 看下一步能够达到哪些点, 如果这些点没有到扩展过 (即没有加入队列), 那么就把这些点加入到队尾。显然, 扩展的格子在队列中是按照到达的最少步数从小到大的顺序排列的。在实现的过程中, 我们需要一个队列存下这些已扩展的格子, 还要用一个数组标记某个格子是否扩展过且到达的最少步数是多少。

算法的伪代码如下:

map——表示棋盘的类型

dis——表示到达棋盘上每个格子的最少步数, 一开始除了起始点, 都为  $-1$

$(sx, sy)$  ——马的起始位置

queue——存储已扩展格子的队列

h——队首指针

t——队尾指针

TourOfHorse ( $sx, sy, map$ )

h ← 0

t ← 1

queue[t] ←  $(sx, sy)$

dis[sx, sy] ←  $-1$

while h < t do

h ← h + 1

$(x, y)$  ← queue[h]

for dir ← 1 to 8 do

$(tx, ty)$  ←  $(x, y)$  在 dir 方向的下一步能够到达的格子

if (map[tx, ty] = '\_' ) and (dis[tx, ty] =  $-1$ ) then



```

dis [tx, ty] ← dis [x, y] + 1
t ← t + 1
queue [t] ← (tx, ty)
return (dis)

```

这种按照达到步数（或者说层次）从小到大扩展的方法我们通常叫做广度优先遍历。

### 3.5 链接的栈和队列

#### 一、链栈的定义与运算

链栈（即链接堆栈）是栈的链接存储表示，或者说它是只允许在表头进行插入和删除运算的单链表，此时单链表的表头指针叫做栈顶指针。一个链栈的示意图如图 3-9 所示，其中 HS 表示栈顶指针。设 HS 的类型为 linklist，数据元素 x 的类型为 elemtype，则在以 HS 为栈顶指针的链栈中，进行栈的各种运算的算法如下：

##### 1. 进栈算法

算法步骤为：

- (1) 为待进栈元素 x 分配一个结点  $p \uparrow$ ，并把 x 赋给  $p \uparrow$  结点的值域；
- (2) 把  $p \uparrow$  结点堆入栈顶。

算法描述为：

```

push (HS, x)
    new (p)
     $p \uparrow . data \leftarrow x$ 

```

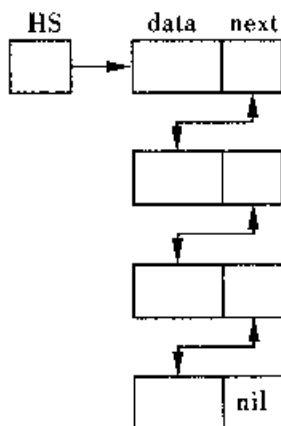


图 3-9 链栈的示意图

```

 $p \uparrow . next \leftarrow HS$ 

```

```

HS ← p

```

##### 2. 出栈算法

假定以函数的形式写出，则算法步骤为：

- (1) 检查 HS 是否为空，若为空则进行“下溢”错误处理；

- (2) 将栈顶结点的值赋给函数名, 并将栈顶指针暂存  $p$ , 以便回收栈顶结点;
- (3) 删除栈顶结点;
- (4) 回收  $p \uparrow$  结点 (即原栈顶结点)。

算法描述为:

```
pop (HS): elemtype;
  if HS = nil then error ( 'underflow' )
  pop  $\leftarrow$  HS  $\uparrow$ . data
  P  $\leftarrow$  HS
  HS  $\leftarrow$  HS  $\uparrow$ . next
  dispose (p)
```

### 3. 读取栈顶元素的算法

此算法很简单, 若不考虑栈空的情况, 只要取出 HS  $\uparrow$ . data 的值即可。

### 4. 置栈空算法

若不考虑回收结点, 则只要将 HS 置空即可。若考虑回收链栈中的所有结点, 则算法如下:

```
setnull (HS);
  while HS < > nil do
    p  $\leftarrow$  HS
    HS  $\leftarrow$  HS  $\uparrow$ . next
    dispose (p)
```

### 5. 判断一个栈是否为空的算法

此算法很简单, 只要当 HS = nil 时返回“真”值, 否则返回“假”值即可。

## 二、链队的定义与运算

链队 (即链接队列) 是队列的链接存储表示, 或者说它是只允许在表尾进行插入和表头进行删除的单链表。一个链队需要队首和队尾两个指针, 其中队首指针  $f$  指向单链表的表头, 队尾指针  $r$  指向单链表的表尾。一个链队的示意图如图 3-10 所示。

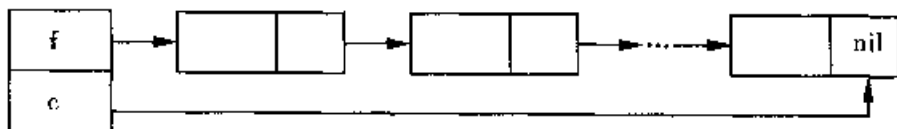


图 3-10 链队的示意图

设  $f$  和  $r$  的类型为 linklist, 则描述  $f$  和  $r$  的结点类型可定义为:

```
type linkqueue = record
  f, r: linklist
end;
```

设 HQ 为具有 linkqueue 类型的一个参数, 它表示一个链队,  $x$  为具有 elemtype 类型的一个参数, 在 HQ 链队中进行插入、删除和置空队运算的算法如下:

#### 1. 插入算法

- (1) 为待入队元素  $x$  分配一个结点  $p \uparrow$ , 并把  $x$  赋给  $p \uparrow$  的值域, nil 赋给  $p \uparrow$  结点的指针域;
- (2) 若链队为空 (即 HQ.f 和 HQ.r 均为空, 检查时判任一个为空即可), 则表明待插入的  $p \uparrow$

结点既是队首结点也是队尾结点,应同时修改队首指针和队尾指针,使之指向  $p \uparrow$  结点,否则把  $p \uparrow$  结点插入队尾,并使队尾指针指向  $p \uparrow$  结点。

算法描述为:

```
insert (HQ, x)
    new_ (p)
    P↑.data ← x
    P↑.next ← nil
    if HQ.r = nil
        then
            HQ.f ← p
            HQ.r ← p
        else
            HQ.r↑.next ← p
            HQ.r ← p
```

## 2. 删除算法

算法步骤为:

- (1) 若链队为空,则进行“下溢”错误处理;
- (2) 把队首结点的值赋给变参  $x$ ;
- (3) 把队首指针暂存指针变量  $p$ ,以便回收该结点;
- (4) 删除队首结点,即若链队中只有一个结点(即  $HQ.f = HQ.r$ ),则应同时把  $HQ.f$  和  $HQ.r$  置为空,否则只修改队首指针,使之指向下一个结点;
- (5) 回收原队首结点(即  $p \uparrow$  结点)。

算法描述为:

```
delete (HQ, X)
    if HQ.f = nil then error ( 'underflow' )
    x ← HQ.f↑.data
    p ← HQ.f
    if HQ.f = HQ.r
        then
            HQ.f ← nil
            HQ.r ← nil
        else
            HQ.f ← HQ.f↑.next
    dispose (p)
```

## 3. 置链队为空的算法

此算法很简单,只要把队首和队尾指针置空即可。

```
setnull (HQ)
    HQ.f ← nil
    HQ.r ← nil
```

不过这样队列中的所有动态结点没有回收,大量浪费了空间,因此可以进行如下改进:

```
setnull (HQ)
p ← HQ.f
while p ≠ HQ.r do
    q ← p ↑ .next
    dispose (p)
    p ← q
    dispose (p)
HQ.f ← nil
HQ.r ← nil
```

### 3.6 小结

这一章学习两种非常重要的运算受限的线性表——栈和队列,它们应用相当广泛,特别是栈在表达式计算和递归中的应用,将来我们在学习搜索算法时深度优先搜索就要使用栈,广度优先搜索就要使用队列,因此读者务必熟练掌握它们的操作和运算的特点。

### 习题三

#### 一、单选题

1. 当利用大小为  $N$  的数组顺序存储一个栈时,假定用  $\text{top} = N$  表示栈空,则向这个栈插入一个元素时,首先应执行( )语句修改  $\text{top}$  的指针。  
A.  $\text{inc}(\text{top})$       B.  $\text{dec}(\text{top})$       C.  $\text{top} = 0$       D.  $\text{top} := 1$
2. 假定一个链栈的栈顶指针用  $\text{top}$  表示,当  $p$  所指向的结点进栈时,执行的操作为( )。  
A.  $p^{\wedge}.\text{next} := \text{top}; \text{top} := p^{\wedge}.\text{next}$   
B.  $\text{top} := p; p^{\wedge}.\text{next} := \text{top}$   
C.  $p^{\wedge}.\text{next} := \text{top}^{\wedge}.\text{next}; \text{top}^{\wedge}.\text{next} := p$   
D.  $p^{\wedge}.\text{next} := \text{top}; \text{top} := p$
3. 假定一个链接的栈顶指针用  $\text{top}$  表示,当进行退栈时所进行的指针操作为( )。  
A.  $\text{top}^{\wedge}.\text{next} := \text{top}$       B.  $\text{top} := \text{top}^{\wedge}.\text{data}$   
C.  $\text{top} := \text{top}^{\wedge}.\text{next}$       D.  $\text{top}^{\wedge}.\text{next} := \text{top}^{\wedge}.\text{next}^{\wedge}.\text{next}$
4. 若让元素 1, 2, 3 依次进栈,每个元素进栈后随时可以出栈,则出栈次序不可能出现( )情况。  
A. 3, 2, 1      B. 2, 1, 3      C. 3, 1, 2      D. 1, 3, 2
5. 在一个顺序队列中,队首指针指向队首元素的( )位置。  
A. 前一个      B. 后一个      C. 当前      D. 后面
6. 从一个顺序队列删除元素时,首先需要( )。



- A. 队首指针循环加 1  
 B. 队首指针循环减 1  
 C. 取出队首指针所指位置上的元素  
 D. 取出队尾指针所指位置上的元素
7. 假定一个不设队列长度变量的顺序队列的队首和队尾指针分别为  $f$  和  $r$ , 则判断队空的条件为( )。

A.  $f+1=r$                       B.  $r+1=f$                       C.  $f=0$                       D.  $f=r$

8. 假定利用数组  $a[N]$  循环顺序存储一个队列, 用  $f$  和  $r$  分别表示队首和队尾指针, 并已知队未滿, 当元素  $x$  进队时所执行的操作为( )。

A.  $a[r \bmod N+1] := x$                       B.  $a[(r+1) \bmod N] := x$   
 C.  $a[r \bmod N-1] := x$                       D.  $a[(r-1) \bmod N] := x$

9. 假定一个带附加头结点的循环链队的队首和队尾指针分别用  $front$  和  $rear$  表示, 则判断队空的条件为( )。

A.  $front = rear$                       B.  $rear = \text{NULL}$   
 C.  $front = \text{NULL}$                       D.  $front = rear$

10. 在一个长度为  $N$  的数组空间中, 顺序存储着一个队列, 该队列的队首和队尾指针分别用  $front$  和  $rear$  表示, 则该队列中的元素个数为( )。

A.  $(rear - front) \bmod N$                       B.  $(rear - front + N) \bmod N$   
 C.  $(rear + N) \bmod N$                       D.  $(front + N) \bmod N$

### 二、算法设计题

- 设计一个递归算法, 返回 1 到  $n$  之间的所有整数平方的和。
- 设计一个递归算法, 把任一十进制正整数转换为  $S$  进制 ( $2 \leq S \leq 9$ ) 数输出。
- 斐波那契 (Fibonacci) 数列的定义为: 它的第一项和第二项分别为 0 和 1, 以后各项为其前两项之和。若斐波那契数列中的第  $n$  项用  $\text{Fib}(n)$  表示, 则计算公式为:

$$\text{Fib}(n) = \begin{cases} n-1 & (n=1 \text{ 或 } 2) \\ \text{Fib}(n-1) + \text{Fib}(n-2) & (n>2) \end{cases}$$

试编写出计算  $\text{Fib}(n)$  的递归算法和非递归算法。

### 三、上机编程题

#### 1. 括号序列。

问题描述:

定义如下规则序列 (字符串):

- (1) 空序列是规则序列;
- (2) 如果  $S$  是规则序列, 那么  $(S)$  和  $[S]$  也是规则序列;
- (3) 如果  $A$  和  $B$  都是规则序列, 那么  $AB$  也是规则序列。

例如, 下面的字符串都是规则序列:

$()$ ,  $[], (())$ ,  $([])$ ,  $()[]$ ,  $() [ () ]$

而以下几个则不是:

$(, [, ], )$ ,  $(, ()), ([ ($

现在, 给你一些由  $'(, ')'$ ,  $'[, ']'$  构成的序列, 你要做的, 是找出一个最短规则序

列,使得给你的那个序列是你给出的规则序列的子列。(对于序列  $a_1, a_2, \dots, a_n$  和序列  $b_1, b_2, \dots, b_m$ , 如果存在一组下标  $1 \leq i_1 < i_2 < \dots < i_n \leq m$ , 使得  $a_j = b_{i_j}$  对一切  $1 \leq j \leq n$  成立, 那么  $a_1, a_2, \dots, a_n$  就叫做  $b_1, b_2, \dots, b_m$  的子列。)

输入:

输入文件仅一行,全部由 '(', ')', '[', ']' 组成,没有其他字符,长度不超过 100。

输出:

输出文件也仅有一行,全部由 '(', ')', '[', ']' 组成,没有其他字符,把你找到的规则序列输出即可。因为规则序列可能不止一个,因此要求输出规则序列中嵌套的层数尽可能地少。

样例:

Bracke. in

( [ ( )

Bracket. out

( ) [ ] ( ) { 最多的嵌套层数为 1, 如层数为 2 时的一种为 ( ) [ ( ) ] }

## 2. 表达式问题。

问题描述:

给出按后缀表示法输入的一个算术表达式,表达式中只有 26 个大写英文字母和加减乘除四个运算符,表达式的长度  $\leq 50$ , 表达式以 # 号结束。编程求出它的等价中缀表达式。

输入:

通过文件输入后缀表达式,文件只有一行,就是后缀表达式。

输出:

输出它的等价中缀表达式。

样例:

输入: INPUT. TXT;

则输出 output. txt;

AB + CD \* EF - \*/

(A + B) / (C \* D \* (E - F))

## 3. 合并果子。

问题描述:

在一个果园里,多多已经将所有的果子打了下来,而且按果子的不同种类分成了不同的堆。多多决定把所有的果子合成一堆。

每一次合并,多多可以把两堆果子合并到一起,消耗的体力等于两堆果子的重量之和。可以看出,所有的果子经过  $n-1$  次合并之后,就只剩下一堆了。多多在合并果子时总共消耗的体力等于每次合并所耗体力之和。

因为还要花大力气把这些果子搬回家,所以多多在合并果子时要尽可能地节省体力。假定每个果子重量都为 1,并且已知果子的种类数和每种果子的数目,你的任务是设计出合并的次序方案,使多多耗费的体力最少,并输出这个最小的体力耗费值。

例如有 3 种果子,数目依次为 1, 2, 9。可以先将 1, 2 堆合并,新堆数目为 3,耗费体力为 3。接着,将新堆与原先的第三堆合并,又得到新的堆,数目为 12,耗费体力为 12。所以多多总共耗费体力  $= 3 + 12 = 15$ 。可以证明 15 为最小的体力耗费值。

输入:

输入文件 fruit.in 包括两行:第一行是一个整数  $n$  ( $1 \leq n \leq 10000$ ), 表示果子的种类数。第二

行包含  $n$  个整数，用空格分隔，第  $i$  个整数  $a_i$  ( $1 \leq a_i \leq 20000$ ) 是第  $i$  种果子的数目。

输出：

输出文件 `fruit.out` 包括一行，这一行只包含一个整数，也就是最小的体力耗费值。输入数据保证这个值小于 231。

样例输入：

3

1 2 9

样例输出：

15

(提示：这里要求使用两个有序队列来编程)



## 4 串

### 4.1 串的基本概念

串（即字符串）是一种特殊的线性表，它的数据元素仅由一个字符组成，计算机非数值处理的对象经常是字符串数据，如在汇编和高级语言的编译程序中，源程序和目标程序都是字符串数据；在学生档案处理中学号、姓名、性别等，一般也作为字符串处理。另外，串还具有自身的特性，常常把一个串作为一个整体来处理。因此，这一章把串作为一个独立结构的概念加以研究，介绍串的存储结构及基本运算。

在早期的程序设计语言中，字符串仅在输入或输出中以直接量的形式出现，并不参与运算。随着计算机的发展与进步，串在文字编辑、词法扫描、符号处理及定理证明等许多领域得到广泛的应用，因而在 pascal 等高级语言中开始引入串变量的概念。如同整型、实型变量一样，所以串变量也可以参加各种运算，而且已归结出一组基本的串的运算。

本节将讨论串的有关概念、表示方法、串的基本运算以及串的应用。

### 4.2 串的定义

字符串（string）是由零个或多个任意字符组成的一个有限的字符序列，一般表示为 ' $a_1 a_2 \dots a_i a_{i+1} \dots a_n$ '。其中的单撇号作为字符串的起止定界，它不属于字符串本身的字符；两个单撇号之间的字符序列称为这个串的值。

串的长度：串中字符的个数  $n$  称为串的长度。

空串：一个串只有 0 个字符，则它被称为空串，长度为 0。

子串：串中连续的任意个字符的子序列称为子串，空串是任意串的子串。

主串：包含子串的串称为主串。

两串相等：对于串  $s_1, s_2$ ，如果  $s_1$  是  $s_2$  的子串，且  $s_2$  是  $s_1$  的子串，则两串  $s_1, s_2$  相等。

串的定义：

`type string = string [maxlen];`



## 4.3 串的实现及基本运算

串的实现:

如何表示一个串呢? 下面介绍几种实现串的数据结构。

### 一、串的数组实现法

顾名思义, 串的数组实现法就是将串作为一个特殊的线性表, 将其用数组表示。需要注意的是: 此时数组的类型为字符类型。因此, 我们可以这样来表示一个串:

```
type
    string = record
        ch: array [1 .. maxlen] of char;
        curlen: integer;
    end;
```

其中 ch 存储的是串的一维数组, 其中每个位置上的字符分别存放在数组的每个下标中; curlen 表示串的当前长度。

在串的数组表示下, 串中的字符都是顺序存储的, 因此这样的表示法特别适合于子串搜索。然而, 用串的这种表示法亦有缺点: 其一是在这种表示法下, 对于串的插入或删除操作都是很复杂的, 要移动许多字符, 因而耗时太多。其二是串的最大长度不能“随即应变”, 必须事先确定最大长度, 这个要求对于串来说不太容易做到, 容易造成最大长度定得太大而浪费许多存储空间, 或最大长度定得太小而在算法执行时产生溢出。

### 二、串的指针实现

类似于串的数组实现, 我们可以将串作为一个特殊的表而用表的指针实现来表示串, 这种方法就是串的指针实现。用指针实现串时, 可以这样表示:

```
type
    tlink = record
        ch: char;
        next: ^tlink;
    end;
    string = ^tlink;
```

在串的链表存储方式下, 对串进行子串的插入或删除操作将会很快, 只要修改相应的指针就可以很快完成。由于是指针实现, 所以对于串的长度没有严格的限制, 在存储空间足够大的时候, 它可以表示任意长度的串。

与串的数组表示法相比, 插入和删除操作之所以很快, 是由于增加了存储空间的代价换来的。如果一个指针域占了 2 个字节的存储空间, 一个字符占用 1 个字节的存储空间, 则指针占用的空间为串中字符所占存储空间的两倍。如果链表使用双向链表, 则需占用更多的存储空间。另外, 由于使用了动态指针, 则访问某个字符的时间则增加了。相比串的数组存储, 串的指针存储比串的数组存储在顺序查找需要更多的时间。

### 三、串的运算

#### 1. 串的连接操作

连接两个串  $s_1$ ,  $s_2$ , 其结果为  $s_c$ .

$s = s_1 + s_2$ ;

例如:

$s_1 = 'ab'$

$s_2 = 'cd'$

$s = s_1 + s_2 = 'abcd'$

也可以使用 `concat` 函数, 即  $s = \text{concat}(s_1, s_2)$

#### 2. 求子串操作

求串  $s_1$  中第  $i$  位开始的, 连续  $j$  个字符组成的子串。

$s = \text{copy}(s_1, i, j)$

例如:

$s_1 = 'abcdefG', i=5, j=2$

$s = \text{copy}(s_1, i, j) = 'ef'$

#### 3. 删除子串操作

删除  $s$  中第  $i$  位开始的连续  $j$  个字符。

`delete(s, i, j);`

例如:

$s = 'abcdefG', i=3, j=3$

则  $s = 'abfG'$

#### 4. 插入子串操作

将  $s_1$  插入  $s$  的第  $i$  位之后。

`insert(s_1, s, i)`

例如:

$s_1 = 'gra', s = 'prom', i=4$

则  $s = 'program'$

#### 5. 求子串的位置

求串  $s_1$  第一次出现在  $s$  的哪个位置。

`result = pos(s_1, s)`

例如:  $s_1 = 'cdea', s_2 = 'ceicdeaacdea'$

则 `result = 5`

#### 6. 求字符串的长度

求字符串  $s$  的长度  $s$ 。

`len = length(s)`

例如:  $s = 'ceicdeaacdeak'$

则 `len = 14`

#### 7. 将数值转换为字符串

`str(value, s)`

把 value 的数值转换成字符串存放在 s 中, value 是一个具有整型或实型的值参数, s 是一个字符串变量。值参数是含有特殊格式命令的表达式。

例如, i 值为 12345, 那么 str (i; 6, s) 使 s 值变为 '12345', 而 x 的值为 3.14E4 时, 那么 str (x; 10; 0, s) 使 s 值变为 '31400'。

#### 8. 将字符串转换为数值

val (st, var, code)

把字符串表达式 s 转换成对应的整型或实型数值 (根据 val 的类型而定), 并把这个值存放在 var 中。s 必须是一个表示数值的字符串, 并符合数值常数的形成规则。首尾空格均不得出现。var 必须是整型或实型变量, code 必须是一个整型变量, 作为过程执行的出错检查代码。如果没有检查出错误, 变量 code 置为 0, 否则置为第一个出错字符位置, 同时转换结果值无意义。

例如: s = '123', 则 val (s, i, result) 将使 i 值变为 123, result 值为 0。如果 s 内含有非数值字符, 例如: '215XQZ', 则 val (s, i, result) 将使 i 值无定义, 且 result 值为 4, 指出错误位置是在第四个字符位置上。若 s 的值为 '215.42E5', 且 x 值是一个实型变量, 那么调用 val (s, x, result) 将使 x 值为 21542000, result 值为 0。

**例题 4-1** 设计一个算法, 判断一个字符串是否为回文串 (回文串即正着读和倒着读相同)。

分析:

我们通过一个例子来分析。输入 S = 'smilelims', 这个字符串的长度为 9, 如果一个串为回文串, 则要求第一个字符等于倒数第一个字符, 第二个和倒数第二个亦要相同, 也就是 S [1] = S [9], S [2] = S [8], S [3] = S [7] ……所以输入 S, 我们将其“倒过来”, 把它的反序记录下来, 再判断这两个串是否相等即可。

**例题 4-2** 你曾经看到过这样的程序吗? 当它运行的时候, 将会输出一些字符, 这些字符恰好组成了这个程序的代码本身。

请你写这样的程序。

请注意, 你的程序运行时将不能访问到源程序 (系统已经将源代码删除)。

分析:

这是一道奇怪的题目, 关键之处在于如何结合字符串与程序的一致性。

以 pascal 为例, 此题要求一个能编译的程序的输出为这个程序本身, 所以我们必须充分利用字符串, 然而需要注意的是在 pascal 语言中需要注意单引号与双引号的问题。

具体程序实现请看程序:

```
program IntrospectiveProgram;
var Stgs: array [1..20] of String;
    i, j: Integer;
begin
  Stgs [ 1 ] := '';
  Stgs [ 2 ] := ' WriteLn ( "program IntrospectiveProgram;" );';
  Stgs [ 3 ] := ' WriteLn ( "var" );';
  Stgs [ 4 ] := ' WriteLn ( " Stgs: array [1..20] of String;" );';
  Stgs [ 5 ] := ' WriteLn ( " i, j: Integer;" );';
  Stgs [ 6 ] := ' WriteLn;';
```

```

Stgs [ 7 ] : = ' WriteLn ( "begin");';
Stgs [ 8 ] : = '';
Stgs [ 9 ] : = ' for i : = 1 to 20 do';
Stgs [ 10 ] : = ' WriteLn ( " Stgs [ ', i; 2, ' ] : = " ', Stgs [ i ], " ";";';
Stgs [ 11 ] : = ' WriteLn;';
Stgs [ 12 ] : = '';
Stgs [ 13 ] : = ' for i : = 1 to 20 do';
Stgs [ 14 ] : = ' for j : = 1 to Length ( Stgs [ i ] ) do';
Stgs [ 15 ] : = ' if Stgs [ i ] [ j ] = Chr ( 34 ) then';
Stgs [ 16 ] : = ' Stgs [ i ] [ j ] : = Chr ( 39 );';
Stgs [ 17 ] : = '';
Stgs [ 18 ] : = ' for i : = 1 to 20 do';
Stgs [ 19 ] : = ' WriteLn ( Stgs [ i ] );';
Stgs [ 20 ] : = ' end. ';
WriteLn ( ' program IntrospectiveProgram;');
WriteLn ( ' var');
WriteLn ( ' Stgs: array [ 1..20 ] of String;');
WriteLn ( ' i, j: Integer;');
WriteLn;
WriteLn ( ' begin');
for i : = 1 to 20 do
    WriteLn ( ' Stgs [ ', i; 2, ' ] : = ' ', Stgs [ i ], ' ' );
WriteLn;
for i : = 1 to 20 do
    for j : = 1 to Length ( Stgs [ i ] ) do
        if Stgs [ i ] [ j ] = Chr ( 34 ) then Stgs [ i ] [ j ] : = Chr ( 39 );
    for i : = 1 to 20 do
        WriteLn ( Stgs [ i ] );
end.

```

## 4.4 串的应用

### 一、串的模式匹配问题

问题的提出:

在实际应用中, 串有个 index 操作被称为“模式匹配”, 这种操作是串的一个独一无二的操作, 这是其他线性结构所不具有的操作。也就是说: 对于两个串 T 和 P, 其中 T 为主串, P 为模式串, 问 T 串是否包含 P 串?

例如, P = 'pascal', T = 'Turbo Pascal 7.0', T 包含 P, P 在 T 的第 6 位首次出现。

将问题抽象化:

给定一个长度为  $m$  的模式串  $P[1..m]$ , 和一个长度为  $n$  的正文  $T[1..n]$ , 找到所有的整数  $s \in [1, n-m+1]$ , 满足: 对于  $\forall x \in [1, m]$  都有  $T[s+x-1] = P[x]$ 。

分析:

### 1. 朴素的模式匹配的算法

朴素模式匹配算法的基本思想是: 从主串  $s$  的第一个字符起和模式  $t$  的第一个字符进行比较, 若相等则进一步比较二者的后续字符, 否则从主串的第二个字符起再重新和模式  $t$  的第一个字符进行比较, 依此类推, 直至模式  $t$  和主串  $s$  中的一个子串相等, 则称匹配成功, 否则称匹配失败。

最朴素的算法是:

算法中变量  $i$  和  $j$  分别指示主串  $s$  和模式  $t$  中当前待比较的字符的位置。该算法在最坏情况下的计算时间复杂度为  $O(mn)$

```
for i := 1 to n do
  for j := 1 to m do
    if i + Lj - 1 ≤ n then
      if T[i..i+Lj] = Pj[1..Lj] then
        begin
          Writeln(i);
          exit;
        end;
```

### 2. KMP 算法

现在我们来讨论由 D. E. Knuth、V. R. Pratt 和 J. H. Morris 提出的一个模式匹配算法, 简称为 KMP 算法。稍后我们将看到 KMP 算法所需的计算时间为  $O(m+n)$ 。由此可知朴素的模式匹配算法不是最优算法。它效率不高的主要原因是没有充分利用在匹配过程中已经得到的部分匹配信息而每次又重头开始比较。KMP 算法正是在这一点上对朴素模式匹配算法作了实质性的改进。在 KMP 算法中, 当出现字符的比较不相等时, 不是像朴素的模式匹配算法那样每次重新比较, 而是利用已经得到的部分匹配的结果, 将模式向右滑动尽可能远的一段距离, 接着继续进行比较。

举个例子: 假如  $T = \text{'tomatogoodtomatobad'}$ ,  $P = \text{'tomatoisbad'}$ , 那么如果按照朴素的匹配算法, 前 7 个字符时第一次匹配失败。按照方法一, 此时会重新从  $T$  的第二位开始比较。其实根据  $P$  的特点, 既然前 6 个字符都匹配成功了, 则应该充分利用, 说明前 6 个字符恰为 'tomato', 由于以 'mato', 'atoi', 开始的串都不可能以  $P$  为前缀, 所以完全没有必要从这些开始比较。而以 'to' 开始的字符串有可能以  $P$  为前缀, 所以下次比较应该直接检测  $T$  的第 5 个字符。

第一趟匹配: tomatogoodtomatobad

↑ 匹配到第六位时失败

tomatoisbad

第二趟匹配: tomatogoodtomatobad

↑ 重新自第 5 位开始匹配, 就可以避免了前几位的重复比较

tomatoisbad

具体地说, 如果匹配到  $P$  的第  $i$  个元素的时候失败, 那么通过刚才的方法计算出一个函数值  $\text{prefix}[i]$ , 然后把  $T$  的当前指针向后移动  $\text{prefix}[i]$  个位置,  $P$  的当前指针不变。这个方法就是



KMP 算法。

$\text{prefix}[i]$  计算的时候实际是自己匹配自己。在匹配的过程中一边利用已经计算出来的  $\text{prefix}$  值来计算新的  $\text{prefix}$  值。具体地说, 假设当前我们已经求出了  $\text{prefix}[1..k-1]$ , 现在我们需要计算  $\text{prefix}[k]$  的值。如果  $\text{st}[\text{prefix}[k-1]+1] = \text{st}[k]$ , 那么显然有  $\text{prefix}[k] = \text{prefix}[k-1]+1$ ; 否则, 如果  $\text{st}[\text{prefix}[\text{prefix}[k-1]]+1] = \text{st}[k]$ , 那么显然有  $\text{prefix}[k] = \text{prefix}[\text{prefix}[k-1]]+1$ ; 否则, 如果  $\text{prefix}[\text{prefix}[\text{prefix}[k-1]]]$  ..... 直到判断是否有  $\text{st}[1] = \text{st}[k]$ , 若  $\text{st}[k] = \text{st}[1]$ , 则  $\text{prefix}[k] = 1$ , 否则  $\text{prefix}[k] = 0$ 。

基本的算法是这样的:

```

Procedure KMP (var p, t: string);
    k := 0; {prefix 函数的计算}
    prefix[1] := 0;
    for i := 2 to m do begin
        while k > 0 and p[k+1] <> p[i] do k := prefix[k];
        if p[k+1] = p[i] then k := k+1;
        prefix[i] := k;
    end;
    q := 0; {利用 prefix 函数计算最大匹配}
    for i := 1 to n do begin
        while q > 0 and P[q+1] <> T[i] do q := prefix[q];
        if p[q+1] = T[i] then q := q+1;
        if q = m then begin
            writeln(i-m);
            exit;
        end;
    end;
end;
    
```

这个程序第一个循环的作用都是, 求自身的  $\text{prefix}$  值, 求的是模式串  $P$  的前缀函数, 这个循环中  $k$  最多移动  $m$  次, 所以时间复杂度为  $O(M)$ ; 而第二段, 则是匹配的过程,  $k$  最多移动  $n$  次, 所以时间复杂度为  $O(N)$ 。

所以总的复杂度为  $O(N+M)$ 。

### 3. 拓展 KMP 算法

问题的提出:

扩展的 KMP 问题:

给定母串  $S$ , 和子串  $T$ 。定义  $n = |S|$ ,  $m = |T|$ ,  $\text{extend}[i] = S[i..n]$  与  $T$  的最长公共前缀长度。

请在线性的时间复杂度内, 求出所有的  $\text{extend}[1..n]$ 。

分析:

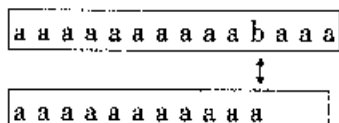
容易发现, 如果有某个位置  $i$  满足  $\text{extend}[i] = m$ , 那么  $T$  就肯定在  $S$  中出现过, 并且进一步知道出现首位置是  $i$ ——而这正是经典的 KMP 问题。

因此可见“扩展的 KMP 问题”是对经典 KMP 问题的一个扩充和加难。



来看一个例子  $S = \text{'aaaaaaaaabaaa'}$ ,  $T = \text{'aaaaaaaaa'}$ 。

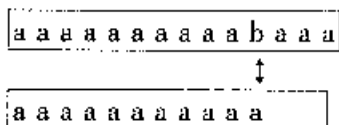
$\text{extend}[1] = 10$



箭头表示失配  
在第 11 个位置失配

这里为了计算  $\text{extend}[1]$ , 我们进行了 11 次比较运算。

然后我们要算  $\text{extend}[2]$ :



箭头表示失配  
在第 11 个位置失配

$\text{extend}[2] = 9$ 。为了计算  $\text{extend}[2]$ , 我们是不是也要进行 10 次比较运算呢? 不然。

因为通过计算得到  $\text{extend}[1] = 10$ , 所以我们可以得到这样的信息:  $S[1..10] = T[1..10] \rightarrow S[2..10] = T[2..10]$ 。

计算  $\text{extend}[2]$  的时候, 实际上是  $S[2]$  开始匹配  $T$ 。因为  $S[2..10] = T[2..10]$ , 所以在匹配的开头阶段是“以  $T[2..10]$  为母串,  $T$  为子串”的匹配。

不妨设辅助函数  $\text{next}[i]$  表示  $T[i..m]$  与  $T$  的最长公共前缀长度。

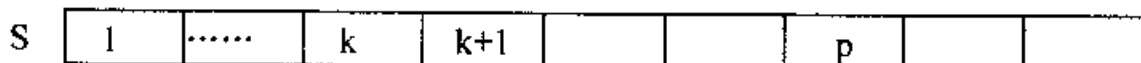
对于这个例子,  $\text{next}[2] = 10$ 。也就是说:

$T[2..11] = T[1..10] \rightarrow T[2..10] = T[1..9] \rightarrow S[2..10] = T[1..9]$ 。

这就是说前 9 位的比较是完全可以避免的! 我们直接从  $S[11] \neq T[10]$  开始比较。这时候一比较就发现失配, 因此  $\text{extend}[2] = 9$ 。

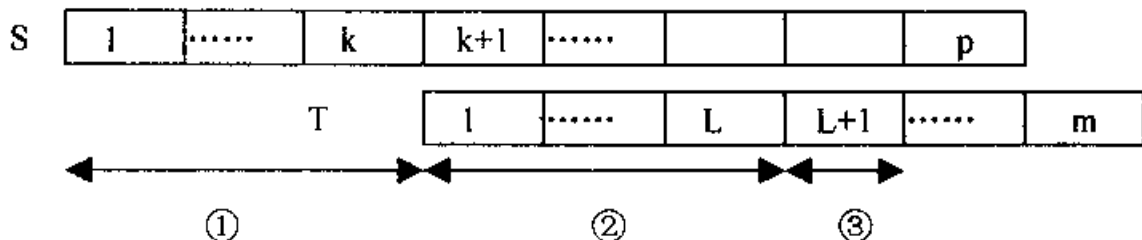
以上的例子是有代表性。下面提出一般的算法。

设  $\text{extend}[1..k]$  已经算好, 并且在以前的匹配过程中到达的最远位置是  $p$ 。最远位置严格地说就是  $i + \text{extend}[i] - 1$  的最大值, 其中  $i = 1, 2, 3, \dots, k$ ; 不妨设这个取最大值的  $i$  是  $a$ 。(下图 1~K 表示已经求出来了  $\text{extend}$  的位置)



根据定义  $S[a..p] = T[1..p-a+1] \rightarrow S[k+1..p] = T[k-a+2..p-a+1]$ , 令  $L = \text{next}[k-a+2]$ 。有两种情况。

第一种情况  $k+L < p$ , 如下图:



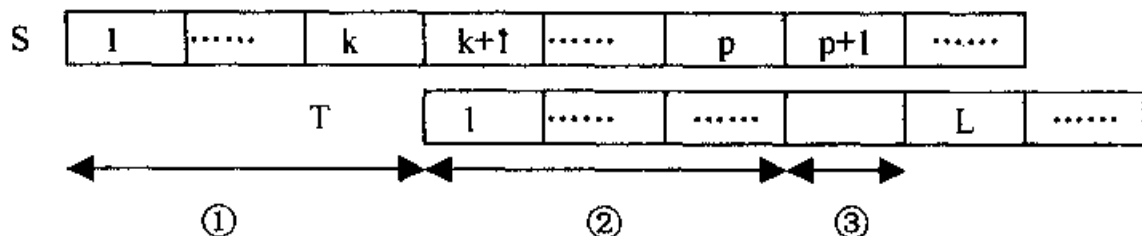
上面的②部分是相等的。③部分肯定不相等, 否则就违反了“ $\text{next}[i]$  表示  $T[i..m]$  与  $T$



的最长公共前缀长度”的定义。(因为  $\text{next}[k-a+2] = L$ , 如果③部分相等的话, 那么就有  $\text{next}[k-a+2] = L+1$  或者更大, 矛盾)

这时候我们无需任何比较就可以知道  $\text{extend}[k+1] = L$ 。同时  $a, p$  的值都保持不变,  $k \leftarrow k+1$ , 继续上述过程。

第二种情况  $k+L \geq p$ 。如下图:



上图的③部分是未知的。因为在计算  $\text{extend}[1..k]$  的时候, 到达过的最远地方是  $p$ , 所以  $p$  以后的位置从未被探访过, 我们也就无从知道③部分是否相等。

这种情况下, 就要从  $S[p+1] \leq T[p-k+1]$  开始匹配, 直到失配为止。匹配完之后, 比较  $\text{extend}[a] + a$  和  $\text{extend}[k+1] + (k+1)$  的大小, 如果后者大, 就更新  $a$ 。

整个算法描述结束。

上面的算法为什么是线性的呢?

很容易看出, 在计算的过程中, 凡是访问过的点, 都不需要重新访问了。一旦比较, 都是从比较以前从不曾探访过的点开始。因此总的时间复杂度是  $O(n+m)$ , 是线性的。

还剩一个问题:  $\text{next}[]$  这个辅助数组怎么计算? 复杂度是多少?

我们发现计算  $\text{next}$  实际上以  $T$  为母串,  $T$  为子串的一个特殊“扩展的 KMP”。用上文介绍的完全相同的算法计算  $\text{next}$  即可。(用  $\text{next}$  本身计算  $\text{next}$ , 具体可以参考标准 KMP 或者作者的程序) 此不赘述。

现在我们给出扩展 KMP 的源代码:

```
procedure extendkmp (s, t: string);
begin
  lens := length (s); lent := length (t);
  s := s + '$'; t := t + '#';
  j := 0;
  while t[1+j] = t[2+j] do j := j + 1;
  next[2] := j; a := 2;
  for i := 3 to lent do begin
    len := next[a] - (i - a); l := next[i - a + 1];
    if len > 1 then next[i] := 1 else begin
      j := max (0, len);
      while t[1+j] = t[i+j] do j := j + 1;
      next[i] := j; a := i;
    end;
  end;
```

```

end;
j := 0;
while s[l + j] = t[l + j] do j := j + 1;
extend[l] := j; a := 1;
for i := 2 to lens do begin
    len := extend[a] - (i - a); l := next[l + (i - a)];
    if l < len then extend[i] := l
    else begin
        j := max(0, len);
        while s[i + j] = t[l + j] do j := j + 1;
        extend[i] := j; a := i;
    end;
end;
end;
end;

```

我们回顾一下扩展的 KMP，它为什么高效？

在计算  $\text{extend}[1..k]$  的时候，已经可以得到一些关于 S 和 T 的信息；在计算  $\text{extend}[k+1]$  的时候，正是充分利用了之前得到的信息，将所有可以避免的比较都避免了，所以最后得到了一个很高效的算法。

再看求最长回文子串。我们很容易提出这样一个算法：枚举回文串的中点，然后向两边扩展。这个算法的复杂度是  $O(n^2)$ ，远远高于  $O(n \log_2 n)$ 。它到底差在哪？

比如  $S = \text{'aaaaaaaaa.....'}$ 。

当以倒数第二个 a 为中点， $\text{'aaaaaaaaa.....'}$ ，实际就是要从  $\text{'aaaaaaaaa.....'}$  中的两个黑色字母开始分别向两边扩展，求最大扩展长度。

当以倒数第三个 a 为中点， $\text{'aaaaaaaaa.....'}$ ，实际就是要从  $\text{'aaaaaaaaa.....'}$  中的两个黑色字母开始分别向两边扩展，求最大扩展长度。

最后归纳一下就是：

$\text{aaaaaaaaa.....}$

对每一个黑色的 a，都要求一次从带下划线的 a 向右、黑色的 a 向左，最多可以扩展多远。因为采用的纯枚举，计算这个的复杂度是  $O(n^2)$ 。

但是我们可以轻松、而有点震惊地发现，这实际上就是一个“扩展的 KMP”问题！母串是黑色部分的逆序串，子串是下划线的 a 及其右边的所有字符。

对于这样一个标准的“扩展 KMP”，该算法采用的实际上是暴力穷举，这样的效率如何能不高！诚如上面的分析，这种暴力穷举等于是放弃了任何已经得到的有用信息。

从另一个角度说，二分法 + 扩展 KMP 算法之所以能够优秀地解决最长回文串问题，也正是因为它减少了计算的冗余，充分利用了已知。

请读者认真领会上面算法的思想，即：已经访问过的点绝不再访问，充分利用已经得到的

信息。

## 二、串的最长回文子串问题

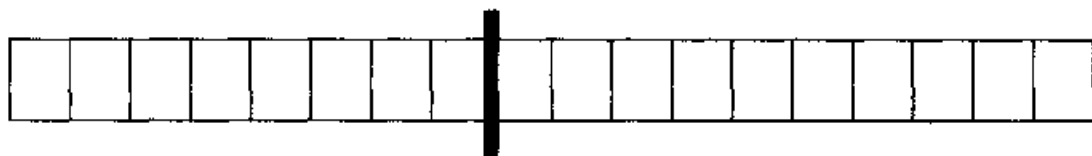
问题：

顺序和逆序读起来完全一样的串叫做回文串。比如 *achca* 是回文串，而 *abc* 不是（*abc* 的顺序为“*abc*”，逆序为“*cba*”，不相同）。

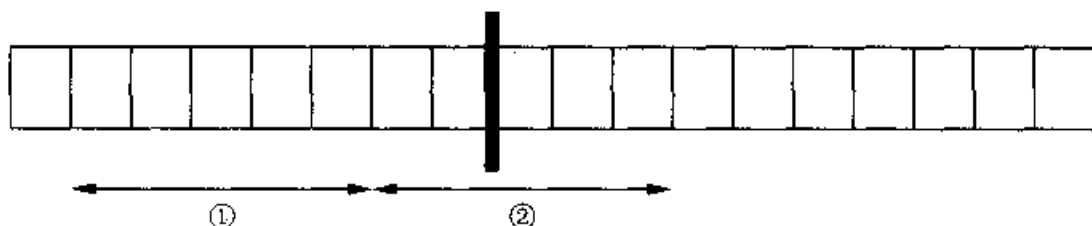
输入长度为  $n$  的串  $S$ ，求它最长回文子串。

分析：

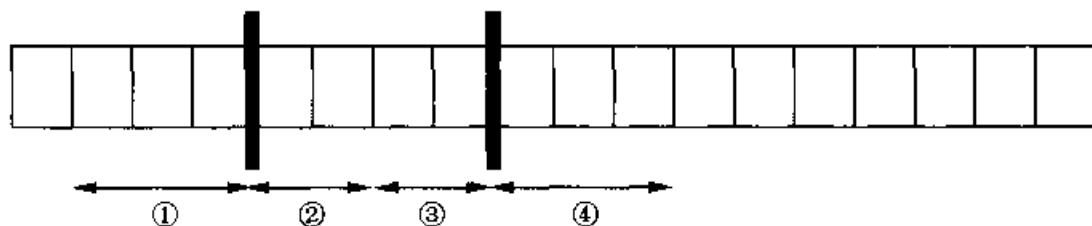
我们把串分成均匀的两部分：



对左边、右边分别递归求最长回文子串，下面的工作只要考虑那些“跨越”粗线的回文串即可。



假设上面是一个回文串，①和②部分对称相等。



如上，实际上就是②和③对称相等、且①和④对称相等。②和①交接的地方（也就是左数第一个竖线），称之为这个回文串的“对称分界点”。

一个回文串必然满足：

(1) 对称分界点到二分点（上图两条粗线条之间的部分）之间，是回文。

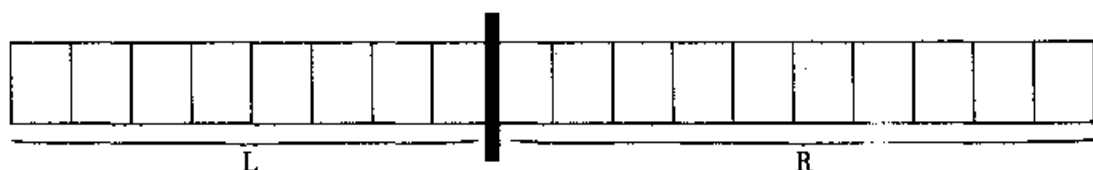
(2) 从对称分界点向左扩展、从二分点向右扩展，必须是完全相等的。

设原串为  $S$ ，左边的串记为  $L$ ，右边的串记为  $R$ 。字符串  $S$  的逆序记为  $r(S)$ 。

用  $\text{extendwest}[i]$  表示第  $i$  个字符向左，和  $R$  匹配，最远可以匹配多远。

显然  $\text{extendwest}[i]$  就是以  $r(L)$  为母串， $R$  为子串的“扩展 KMP”。 $O(n)$  内可以解决。

类似的  $\text{extendeast}[i]$  表示从第  $i$  个字符向右扩展，和  $r(L)$  匹配，最远可以匹配多远。



显然  $\text{extendeast}[i]$  是以  $L$  为母串,  $r(L)$  为子串的“扩展 KMP”。线性时间内可以解决。求出来  $\text{extendwest}$  和  $\text{extendeast}$  有什么用呢?

我们枚举“对称分界点”, 设为  $A$ ; 设二分点为  $M$ 。根据条件, 只要满足:

$$\text{extendeast}[A] * 2 \geq M - A + 1$$

就肯定存在以  $A$  为对称分界点的回文串。 $S[A..M]$  称为基本回文串。

因为要求长度最大, 我们在基本回文串的基础上向两边扩展, 容易发现扩展的长度就是  $\text{extendwest}[A-1]$  (规定  $\text{extendwest}[0] = 0$ ), 所以此时的长度:

$$\text{LENGTH} = \text{extendwest}[A-1] * 2 + M - A + 1$$

枚举所有可能的“对称分界点”(总共不超过  $n$  个), 对每个分界点, 根据上面的分析, 只要用  $O(1)$  的时间复杂度就能判定它的合法性, 以及求出以该点为分界点时的最大回文串长度。最后取最大值即可。

注意到上面的讨论中, 回文串的“重心”在  $L$  中——所谓重心就是回文串中点。所以“对称分界点”也在  $L$  中。实际上还有可能是下面的情况:

类似处理即可, 此不赘述。

以上我们就在  $O(n)$  的时间复杂度内 ( $n = |S|$ ), 求出了跨越“二分点”的最长回文串长度。

分析一下时间复杂度。设计算长度为  $n$  的串的时间复杂度是  $f(n)$ , 一个粗略的递推可以写成:

$f(n) = 2f(n/2) + n$  (其中  $n$  是求跨越二分点的最长回文子串的复杂度,  $f(n/2)$  分别是递归处理两边的复杂度)

$$f(1) = 1$$

很容易算出:

$$f(n) \sim n \log_2 n$$

也就是说该算法复杂度是  $O(n \log_2 n)$ 。

### 三、串的最长重复子串问题

问题的提出:

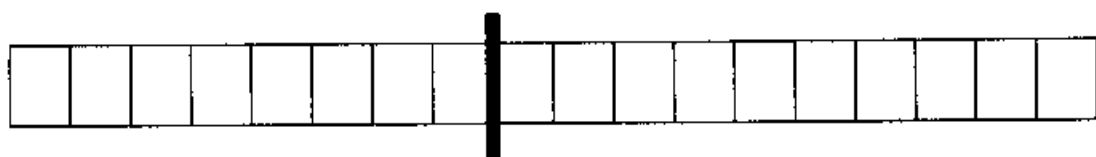
如果一个串  $x$  在  $S$  中出现, 并且  $xx$  也在  $S$  中出现, 那么  $x$  就叫做  $S$  的重复子串。

输入长度为  $n$  的串  $S$ , 求它的最长重复子串。

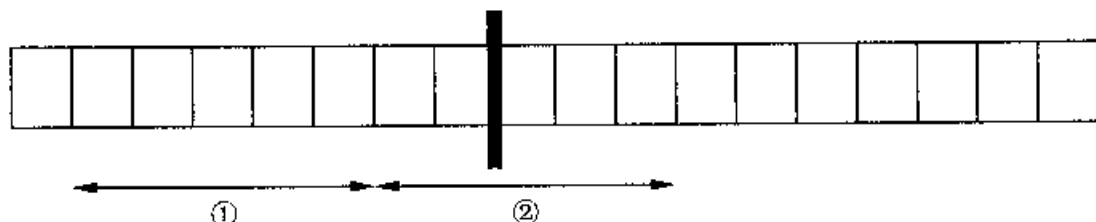
分析:

有了最长回文串的解题基础, 研究最长重复子串就要简单多了。

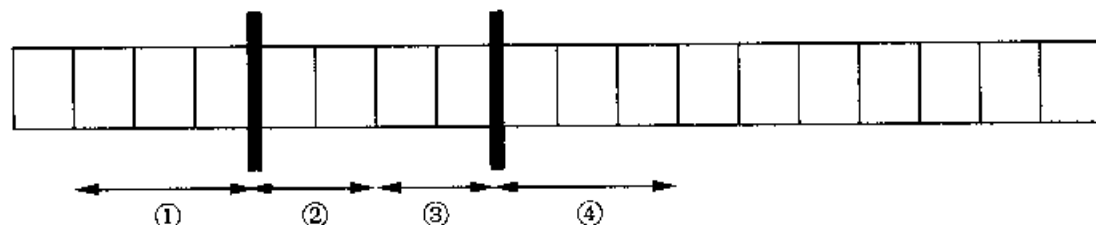
我们把串分成均匀的两部分:



对左边、右边分别递归求最长重复子串，下面的工作只要考虑那些“跨越”粗线的重复子串即可。



假设①和②部分完全相等。(也就是说存在一个长度为5的重复子串)



如上，实际上就是②和③相等，且①和④相等。①和②交接的地方（也就是左边的竖线），称之为这个重复子串的“重复分界点”。（重复分界点到二分点的距离，实际上就是重复子串的长度）

设原串为  $S$ ，左边的串记为  $L$ ，右边的串记为  $R$ 。字符串  $S$  的逆序记为  $r(S)$ 。

用  $\text{extendeast}[i]$  表示第  $i$  个字符向右，和  $R$  匹配，最远可以匹配多远。

显然  $\text{extendeast}[i]$  就是以  $L$  为母串， $R$  为子串的“扩展 KMP”。 $O(n)$  内可以解决。

类似的  $\text{extendwest}[i]$  表示从第  $i$  个字符向左扩展，和  $r(L)$  匹配，最远可以匹配多远。

显然  $\text{extendwest}[i]$  是以  $r(L)$  为母串， $r(L)$  为子串的“扩展 KMP”。 $O(n)$  内可以解决。

求出来  $\text{extendwest}$  和  $\text{extendeast}$  有什么用呢？

我们枚举“重复分界点”，设为  $A$ ；设二分点为  $M$ 。根据条件，只要满足：

$$\text{extendeast}[A] + \text{extendwest}[A-1] \geq M - A + 1$$

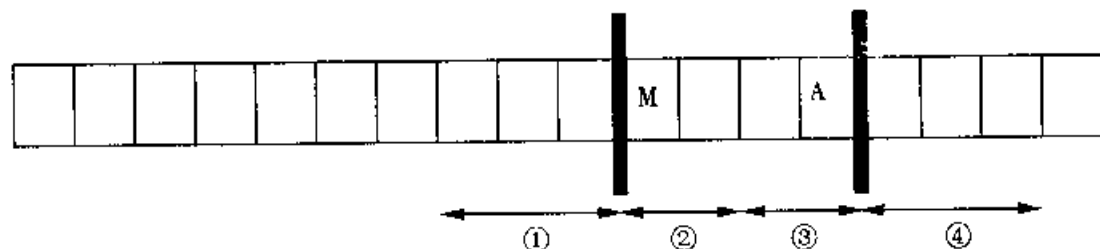
就肯定存在以  $A$  为重复分界点的重复子串。此时的串长度是

$$\text{LENGTH} = M - A + 1$$

枚举所有可能的“重复分界点”（总共不超过  $n$  个），对每个分界点，根据上面的分析，只要用  $O(1)$  的时间复杂度就能判定它的合法性以及求出以该点为分界点时的最大重复串长度。

最后取最大值即可。

注意到上面的讨论中，重复子串是“偏左”的，实际上还有可能“偏右”，如下：



类似处理即可，此不赘述。

至此问题2——最长重复子串——也解决了。时间复杂度和第一个问题相同，也是  $O(n \log_2 n)$ 。

#### 四、串的同构问题

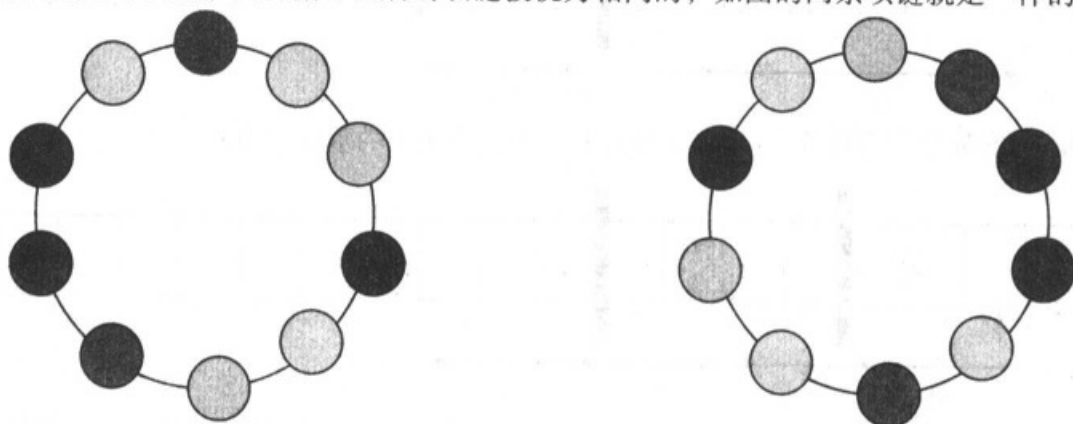
问题的提出：

有两条环状的项链，每条项链上各有  $N$  个多种颜色的珍珠，相同颜色的珍珠，被视为相同。

问题：判断两条项链是否相同。

分析：

由于项链是环状的，因此循环以后的项链被视为相同的，如图的两条项链就是一样的。

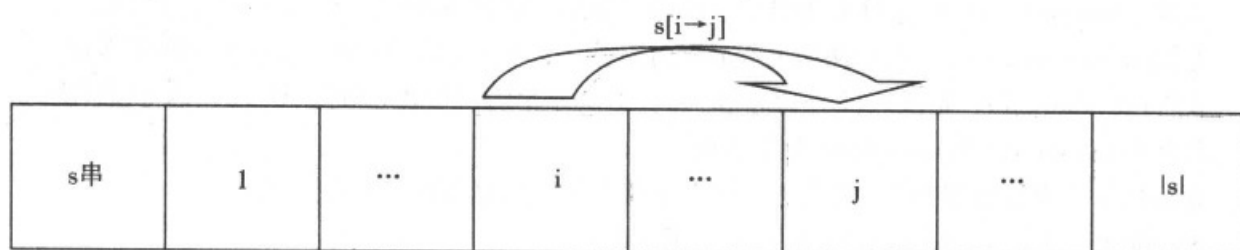


为了方便解题，我们定义一些概念：

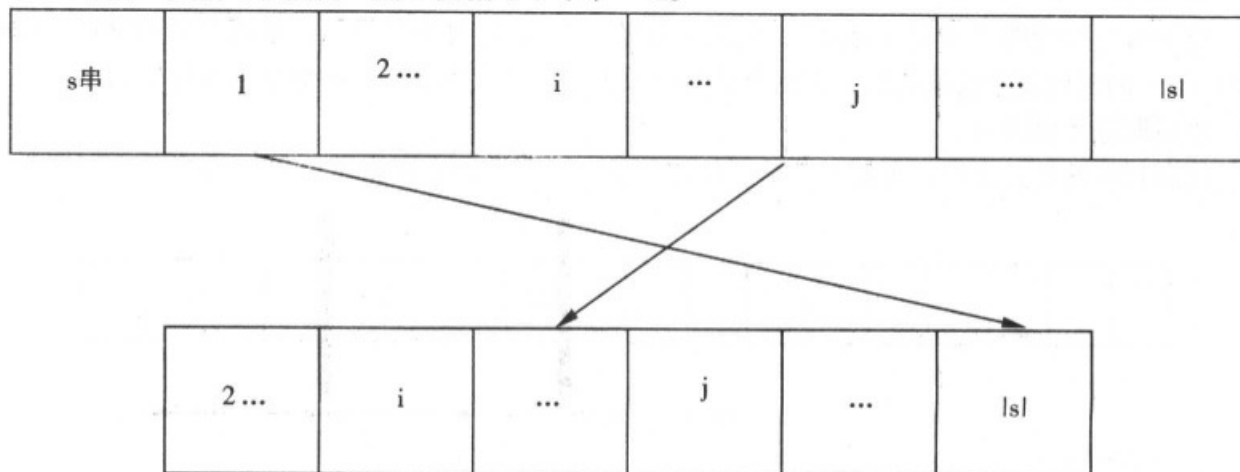
(1)  $|s| = \text{length}(s)$ ，即  $s$  的长度。

(2)  $s[i]$  为  $s$  的第  $i$  个字符。

(3)  $s[i \rightarrow j] = \text{copy}(s, i, j - i + 1)$ 。这里  $1 \leq i \leq j \leq |s|$ 。



(4) 定义  $s$  的一次循环  $s(1) = s[2 \rightarrow |s|] + s[1]$ ； $s$  的  $k$  次循环 ( $k > 1$ )， $s(k)$  为  $s(k-1)$  的一次循环；另外  $s$  的 0 次循环  $s(0) = s$ 。



(5) 如果字符串  $s_1$  可以经过有限次循环得到  $s_2$ , 则称  $s_1$  和  $s_2$  是循环同构的。

(6) 设有两个映射  $f_1, f_2: A \rightarrow A$ , 定义  $f_1$  和  $f_2$  的连接  $f_1 \circ f_2 (x) = f_1 (f_2 (x))$ 。

问题的数学描述: 给定两个长度相等的字符串,  $|s_1| = |s_2|$ , 判断它们是否是循环同构。

#### 1. 朴素的算法

易知,  $s_1$  的不同的循环串最多只有  $|s_1|$  个, 即  $s_1, s_1^{(1)}, s_1^{(2)}, \dots, s_1^{(|s_1|-1)}$ 。所以只需要把他们一一枚举, 然后分别与  $s_2$  比较即可。

这个算法的时间复杂度为  $O(N^2)$ 。( $N = |s_1| = |s_2|$ )

当然, 如果  $N$  的范围很小, 则此算法很好, 思维简单, 易于实现。

可是, 如果  $N$  的范围在扩大一些呢? 此算法将无法胜任。

#### 2. 匹配算法

我们重头分析这道题目, 首先构造新的模型:  $S = s_1 + s_1$  为主串,  $s_2$  为模式串。如果  $s_1$  和  $s_2$  是循环同构的, 那么  $s_2$  就一定可以在  $S$  中找到匹配!

在  $S$  中寻找  $s_2$  的匹配是有很多  $O(N)$  级的算法的。本题最优算法的时空复杂度均为  $O(N)$  级。这已经是理论的下界了, 因为输入的复杂度已为  $O(N)$ 。

比较上述两个算法: 通过算法的执行过程, 我们找到了算法的实质: 模式匹配。

最后通过模型的转换, 使得可以直接套用模式匹配的 KMP 算法, 从而得到  $O(N)$  级别的算法。

#### 例题 4-3 病毒的 DNA 问题。

问题描述:

有一种奇特的病毒, 它的 DNA 序列是环状的, 而一般的生物的 DNA 都是线状的, 且由科学家发现: 生物被此种病毒侵袭的可能性与生物和病毒的 DNA 序列最大公共长度有关, 由于病毒是环状的, 所以它可以循环重复地匹配。科学家们经过大量的试验发现: 如果生物和病毒 DNA 序列的最大公共部分的长度还没有病毒的 DNA 长, 病毒是无法安身的, 也就是说这个生物被侵染的几率是 0; 否则, 最长公共部分的长度和被侵染的几率满足下面的关系式: 生物被侵染几率 = 最大公共部分长度 / 生物 DNA 长度。

现在已知病毒的 DNA 序列和某生物的 DNA 序列, 你必须求出此生物被侵染的几率是多少。

分析:

设  $A$  为病毒的环状 DNA 字符串,  $A$  的长度为  $N$ 。设  $B$  为生物的线状 DNA 字符串,  $B$  的长度为  $M$ 。那么题目所求: 环串  $A$  和线串  $B$  的最大可循环公共子串长度。

此题实际上比较类似一般字符匹配问题, 不同点在于此题有环串存在!

经过初步分析, 很容易想到用动态规划来解此类求公共最大长度的题目, 而且稍加分析就可设计相应的动态规划: 设  $f[i, j]$  表示以线串  $B$  的第  $i$  位和环串  $A$  的第  $j$  位结尾的最大公共子串的长度。

动态转移方程为:

$$f[i, j] = \begin{cases} f[i-1, j-1] + 1, & A[j] = B[i] \text{ 且 } i < j \leq n \\ f[i-1, n] + 1, & A[j] = B[i] \text{ 且 } j = 1 \\ 0 & A[j] \neq B[i] \end{cases}$$

最后的答案为:

$Ans = \max (f[i, j], 1 \leq i \leq m, 1 \leq j \leq n)$

可是, 这种方法的时间复杂度为  $O(N * M)$ , 还可以进一步优化。

经过分析, 不必求出依次所有的  $f[i, j]$ , 只有当  $B[i] = A[j]$  时, 才有必要求  $f[i, j]$ , 其余的  $f$  值全为 0。

又因为  $A, B$  中的字符只有  $['a' \dots 'z', 'A' \dots 'Z']$ , 那么只需在开始时用链表记录  $'a' \dots 'z', 'A' \dots 'Z'$  出现的位置, 动态规划的过程中就可以实现这个优化。

然而, 局部的优化并不能使得问题的复杂度降低, 我们得重新分析问题。

我们发现, 动态规划未用到另一条件: 只有最大公共子串的长度大于等于  $N$  时, 才有必要计算这个长度。

顺着这个思路我们可以得出更加优秀的解法, 请读者自行思考。

#### 例题 4-4 串的染色问题。

问题描述:

$N \times M$  的方阵上, 被指定了一些格子, 请你用一些不相交的正方形将这些方格完全覆盖。为了清楚地表示一种覆盖方案, 我们用大写字母给各正方形“染色”, 要求相邻(四连通)的正方形所涂颜色各不相同。

你的任务就是求出所有覆盖方案中字典序最小的一个(将方阵染色后的各行依次连接成为一个字符串, 进而比较大小)。

输入

第一行包含两个整数,  $M$  和  $N$ 。( $1 \leq M \leq 100, 1 \leq N \leq 80$ )

接下来  $M$  行, 每行  $N$  个字符, 是 '?' 或 '.'。如果第  $i$  行第  $j$  列的字符是 '?', 那么表示该方格需要被覆盖; 如果是 '.', 则表示这个方格没有被选中。

输出

$M$  行  $N$  列的字符方阵, 为字典序最小的覆盖方案。其中不需要被覆盖的格子用 '.' 表示, 需要被覆盖的格子, 则用大写字母表示其颜色。

样例输入	样例输出
5 5	AAAB.
????.	AAA. A
??? .?	AAABB
?????	BBCBB
?????	BBAC.
????.	

分析:

首先需要明确, 无论当前未被覆盖格子组成什么形状, 剩下的格子一定可以被若干正方形覆盖的。这个是很明显的, 每一个小格子都可以被看作一个  $1 \times 1$  的正方形。至于使用颜色种类的限制, 26 种颜色是比较多的, 可以暂且不考虑这个问题。

这样也就保证了, 如果图中一部分的格子已经被涂上颜色, 那么无论具体情况如何, 剩下的



格子总存在着覆盖的方案。

由于本题的关键在于“字典序最小”，也就是说越靠前的方格所填字母应当尽可能小。因此很容易想到，从左上角的格子开始，依次从左到右、从上到下，一行一行地将所有选定方格全部覆盖。覆盖时要注意尽量给靠前的格子选择较小的字母。

但每次找到覆盖一个格子的时候，需要确定的有两个因素：用什么字母、用多大的正方形。这两个因素看上去是互相制约的，并不好确定：正方形的大小决定了相邻的部分，也就决定了在选择字母时的限制条件；而使用的字母也决定了正方形不能和一部分同色正方形相邻，进而限制了正方形的大小。

因此，要想确定覆盖方案，先要明确两个因素的先后关系。根据前面分析中提到的，当前将要被覆盖的格子一定是所有未被覆盖的格子中，最上面一行中最靠左边的一个，也就是在比较方案大小时连成的字符串中位置最靠前的一个。因此，只要它尽量小就可以了。虽然这样可能会限制这个正方形的大小，但这样得出的方案已经比其他方案的字典序小了，也就一定能够得出最后的最优解。

所以，处理一个待覆盖的格子时，我们先收集一下与其相邻的、已染色的格子的颜色，并将这些颜色从可选的颜色中去掉。而这个待覆盖的格子的颜色，也就是剩下的可选颜色中最小的一个。

确定了待覆盖格子的颜色，那么接下来需要解决的问题就是这个正方形的大小了。我们可以让正方形不断地扩大，每次在右边和底部分别加上一列和一行，直到无法扩大为止。扩大是需要满足一些条件的：

首先，加入的部分一定要全部是被选出需要覆盖的格子；而且在前面的处理中，都没有被覆盖过。其次，与这些新加入的格子相邻的，不能有与当前确定的正方形颜色相同的格子。最后，被加入的格子中最上面的一个，也就是即将成为新的正方形右上角的格子，应该无法选择比当前确定的字母更小的字母。

这样，我们不断地扩大正方形，直到上面的条件不能够被满足，也就确定了当前需要使用的正方形的大小和颜色。将这片区域染色以后，就可以继续找最靠前的待覆盖格子进行相同的处理了。

## 4.5 小结

串的处理是计算机非数值处理的重要部分，充分体现了计算机应用之广泛。在串的应用一节中重点讨论了几个很重要的问题，提出解决的几种方案，特别是优化方面，值得大家仔细去研读，并且多数算法没有给出程序具体编码，希望读者自己编程实现，这对掌握串的应用是大有好处的。

## 习题四

### 一、选择题

- 下列哪个(些)字符串是字符串 'aaabababbabba' 和 'bbbabba' 的最长公共子串( )。  
A. abba                      B. abbba                      C. bbab                      D. habba
- 字符串 abaabaaab 的 next 函数的 next [3] 的值为( )。  
A. 3                          B. 4                          C. 5                          D. 6
- 字符串 'ababaaahbababbaabab' 的最长回文子串为( )。  
A. aabbabababbaa                      B. ababa  
C. aabbaba                      D. abababbaaba
- 在字符串 'abababbabababab' 的所有子串中出现次数最多的字符串是( )。  
A. ab                          B. ba                          C. a                          D. b
- 字符串 'aaaabbbbbaaaaa' 和 'aaaaaaaaa' 的最长公共后缀是( )。  
A. aaaabbbbbaaaaa                      B. aaaaaaaaaa  
C. aaaa                          D. aaaaa
- 长度为 6, 只由 'a', 'b' 构成的字符串有多少种( )。  
A. 6                          B. 12                          C. 24                          D. 64
- 字符串 'aaab' 和 'aabbba' 比较大小是( )。  
A. >                          B. <                          C. =                          D. 不确定
- 字符串 'adfafgaohgpapagG' 的长度是( )。  
A. 17                          B. 16                          C. 15                          D. 14
- 运算 'aaa' + 'bbb' 的结果是( )。  
A. ab                          B. ba                          C. aaabbb                      D. ababab
- 字符 'aABaa' 的最长回文子串是( )。  
A. aABaa                      B. aabaa                      C. aa                          D. B

### 二、阅读程序写结果

1.

var

str : string;

len, i, j : integer;

nchr : array [0..25] of integer;

mmin : char;

begin

mmin := 'z';

readln (str);

len := length (str);

```

i := len;
while i ≥ 2 do begin
    if str[i - 1] < str[i] then break;
    dec(i);
end;
if i = 1 then begin
    writeln('No result!');
    exit;
end;
for j := 1 to i - 2 do write(str[j]);
fillchar(nchr, sizeof(nchr), 0);
for j := i to len do begin
    if (str[j] > str[i - 1]) and (str[j] < mmin) then
        mmin := str[j];
    inc(nchr[ord(str[j]) - ord('a')]);
end;
dec(nchr[ord(mmin) - ord('a')]);
inc(nchr[ord(str[i - 1]) - ord('a')]);
write(mmin);
for i := 0 to 25 do
    for j := 1 to nchr[i] do
        write(chr(i + ord('a')));
writeln;
end.
输入: zzyzcccbbaaa
输出:

```

2.

```

program Gxpl;
var i, n, jr, jw, jb: integer;
    chl: char;
    ch: array[1..20] of char;
begin
    readln(n);
    for i := 1 to n do read(ch[i]);
    jr := 1; jw := n; jb := n;
    while (jr ≤ jw) do
        begin
            if (ch[jw] = 'R')

```

```

        then begin
            ch1 := ch[jr]; ch[jr] := ch[jw]; ch[jw] := ch1; jr := jr + 1;
        end
    else if ch[jw] = 'W'
    then jw := jw - 1;
    else begin
        ch1 := ch[jw]; ch[jw] := ch[jb]; ch[jb] := ch1; jw := jw - 1;
        jb := jb - 1;
    end
end;
for i := 1 to n do write(ch[i]);
writeln;
end.
输入: RWWRRWWR

```

### 三、程序填空

给定一个 01 串, 请你找出长度介于 A、b 之间, 重复出现次数最多的 01 串。

输入: A、b ( $0 < a \leq b \leq 12$ )

由 0、1 组成的序列, 由 '.' 结尾。

输出: 要求的串。

program timu;

```

var i, j, s, k, a, b, max; integer;
    m: array [1..8192] of integer;
    two, v: array [1..20] of integer;
    c: char;

```

begin

```

    for i := 1 to 15 do _____
    readln(a, b);
    read(c);
    s := 1; k := 1;
    while c <> '.' do begin
        s := s shl 1 + ord(c) - 48;
        if _____ then s := (s - two[b + 1]) mod two[b] + two[b];
        inc(m[s]);
        if k < b then
            for i := a to k - 1 do _____
            inc(k);

```

```

    read (c);
end;
for i : = two [b] to two [b + 1] do
    if m [i] > 0 then
        for j : = a to b - 1 do
            m [ (i mod two [j]) + two [j]] : = _____
        max : = 0;
    for i : = two [a] to two [b + 1] do
        if m [i] > max then max : = m [i];
    for i : = two [a] to two [b + 1] do
        if m [i] = max then begin
            j : = 0; k : = i;
            repeat
                j : = j + 1;
                v [j] : = k and 1;
                k : = k shr 1;
            until _____
            while j > 0 do begin write (v [j]); dec (j); end;
            writeln;
        end;
    end.

```

#### 四、上机编程题

1. 通过键盘分别读入两个位数不超过 200 的正整数，编程计算并输出前一个整数减去后一个整数的差。

例如：输入：12 34567890  
16789000

则输出：1117778890

2. 编程对一个只含有大小写英文字母、逗号、单引号、问号及空格的句子（测试句子由键盘输入，句子以“.”结束，其长度不超过 200 个字符）分别进行如下处理：

①把所有的大写字母转换成小写字母；

②去掉多余的空格（只保留一个）；

③对连续的字母（不包括标点符号和空格）要进行压缩，压缩的办法是先存入该字母，再在它之后存入一个数字表示它重复个数（连续字母不会超过 9 个），例如 HhHhh 转换成 h5；

请分别输出三种处理结果。

3. 给定一个  $n$  位的正整数  $x$ ，将它的各位数字重新排列，得到一个大于  $x$  的最小的回文数，如果没有则输出 0， $x$  的首位不为 0。  $N \leq 1000$ 。（提示：对于题目中间要求大于  $x$  的最小的，可以考虑一位一位确定）

4. 小 S 终于有了自己的电子邮箱！她决定写 E-mail 告诉自己所有的好朋友。邮件很快就写好了，可是……小 S 觉得这封邮件每行长度并不一致，很不好看。于是，她想请你帮忙给她的电子邮件重新排版。使得排版后，整段文字中每行的长度都等于给定的宽度（包括最后一行）。

为了使整段文字中每一行的长度都相同，我们可以在单词之间加入一些空格。看下面一段话（第一行的星号表示应有的宽度）：

\*\*\*\*\*

There is an electric cooker  
in the kitchen.

如果直接在单词之间加入空格，我们可以得到：

\*\*\*\*\*

There is an electric cooker  
in the kitchen.

这样排版虽然整齐了，但看起来依然不很美观，如果我们将第一行的“cooker”调到第二行，再加入一些空格，就可以得到下面的效果，比第一种方案要美观一些：

\*\*\*\*\*

There is an electric  
cooker in the kitchen.

为了量化表示一个排版效果的好坏，我们对于单词间的间隙定义一个权值  $B$ ，一个含有  $t$  个空格的间隙的权值  $B$  就等于  $(t-1)^2$ 。对一种排版方案的总评价就是这篇文章中所有间隙的权值之和。我们不妨认为，总评价的数值越低，这种排版方案越美观。

上面的例子中，第一种排版方案的总评价为  $1+6^2+7^2=86$ ，第二种排版方案的总评价为  $2^2+3^2+3^2+2^2+2^2=34$ 。

在你给出的排版方案中，每一行都应该是由单词开头和结尾的。也就是说每一行的开头和结尾都不能出现空格。一般情况下，每一行的长度都应该等于给定的宽度。除了下面这种特殊情况：

某一行只含有一个单词，而这个单词的长度又不到给定的宽度。这时，该单词应该出现在这一行的开头，并且结尾没有空格。我们将忽略它与下一个单词之间的间隙，但对于每一个这种情况，将给总评价加上 500。

任务：

对于给出的一段文字，通过在单词间加入空格，使得每行的长度都达到已知宽度。并且要求得到的排版方案总评价尽量低。

输入：

第一行是一个整数  $W$  ( $1 \leq W \leq 80$ )，表示指定的宽度。

接下来的一行或几行给出了这段话。整段话中的单词会被回车和空格隔开。（单词由 ASCII 码在 33~126 之间的字符组成。）

输入保证，每个单词的长度都不会超过  $W$ ，且所有单词的总长度不会超过 10000。

输出：

一段文字，不包含空行，表示最优排版方案。

如果有多个最优方案，用以下方法判断：对于两个有同样总评价的方案，找出它们之间第一处不同的间隙，不要输出间隙较大的一个方案。

## 5 数组、特殊矩阵和广义表

### 5.1 多维数组

#### 一、数组的逻辑结构

数组是一种很常见的数据结构，它可以看作是线性表的一种推广形式，比如一维数组就是一个线性表。但正由于数组的元素本身可以是某种具有特定结构的数据，所以一个二维数组可以看作是一个数组元素是“一维数组”的一维数组，一个三维数组可以看作是一个数组元素是“二维数组”的一维数组，一个  $n$  维数组可以看作是一个数组元素是“ $(n-1)$  维数组”的一维数组。

数组是有着固定格式与数量的数据结构，也就是说数组一旦被规定，那么它的每一维的大小及上下界都不能被改变，且数组不能再作插入或删除元素的操作。另外，每一个数组元素都是用唯一下标来标示的。我们通常在数组上执行以下两种操作：

- (a) 取值操作：即读取一个给定数组下标作对应的元素；
- (b) 赋值操作：即存储或修改一个给定数组下标作对应的元素。

图 5-1 即一个二维数组  $A[1..2][1..3]$  的逻辑状态：

$A_{11}$	$A_{12}$	$A_{13}$
$A_{21}$	$A_{22}$	$A_{23}$

图 5-1 示例图

#### 二、数组的内存映像

由于内存的地址空间是一维的，所以一个多维数组就必须要有个固定的方式使它的每一个元素能根据下标在内存中对应一个相对固定的存储地址。我们通常是通过一个映像函数来获得下标所对应的存储地址。

对于一维数组，我们直接按照下标顺序分配即可。

对于多维数组，为了把一个多维下标映射成一维下标，通常有两种形式：

- (a) 按照行优先的顺序（先行后列）存储的，如 BASIC、PASCAL、C。
- (b) 按照列优先的顺序（先列后行）存储的，如 FORTRAN。

行优先的规律是：最右边的下标先从小到大变化，循环一遍后，右边第二个下标再变化……列优先则正好相反：最左边的下标先从小到大变化，循环一遍后，左边第二个下标再变化……

对图 5-1 所给的二维数组，按照行优先顺序和列优先顺序的内存映像如图 5-2 的 (a) 和 (b) 所示。

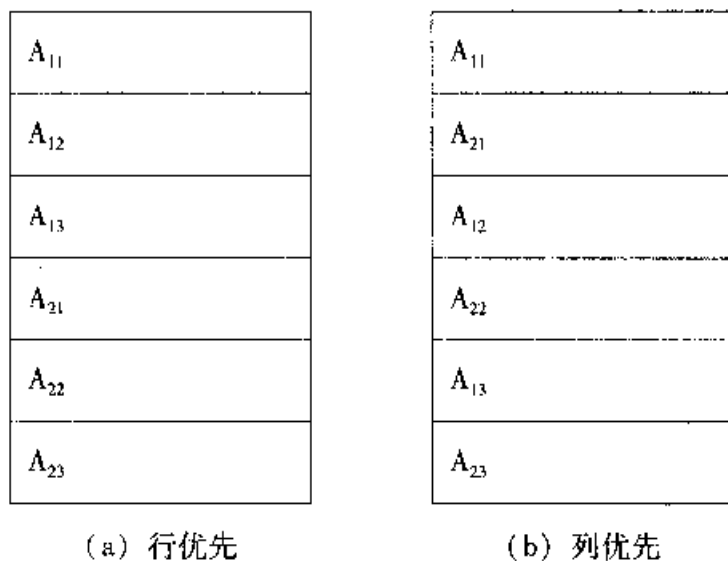


图 5-2 存储映像示意图

正由于多维数组的存储方式的有序性，所以我们一旦知道了数组的基地址，就可以求出数组每一个元素的存储地址。

这里以行优先存储顺序的二维数组  $A_{mn}$  为例，如果每个数组元素占  $p$  个存储单元，基地址为  $POS(A_{11})$ ，对于  $A_{ij}$  我们有：

$$POS(A_{ij}) = POS(A_{11}) + ((i-1) * n + j-1) * p$$

这是因为  $A_{ij}$  之前有  $i-1$  行，每一列由  $n$  个元素，而第  $i$  行中有  $j-1$  个数组元素在它之前。

推广到 3 维数组  $A[b \cdots c][d \cdots e][f \cdots g]$ ， $A_{ijk}$  的存储地址是：

$$POS(A_{ijk}) = POS(A_{bdf}) + ((i-b) * (e-d+1) * (g-f+1) + (j-d) * (g-f+1) + k-1) * p$$

## 5.2 稀疏矩阵

在计算机中，存储矩阵的一般方法是采用二维数组，这样可以随机地访问一个元素，较容易实现矩阵的各种运算。但当一个矩阵中有用的元素个数远远少于无用元素个数时（即稀疏矩阵），这种存储方法既浪费了大量空间存储无用元素，在运算时又浪费了大量时间来访问无用元素。显然是不可取的，那么可以考虑存储少数有用元素。

### 一、稀疏矩阵的三元组存储

对于稀疏矩阵的每一个有用元素，可以用它所在的行号、列号以及元素值这样一个三元组来表示。这样如果矩阵  $A$  中有  $k$  个有用元素，则得到了  $k$  个三元组，用一个线性表存储起来，就得到了一个稀疏矩阵的三元组线性表。

例如下面就是一个稀疏矩阵以及其对应的线性表：



0	0	0	0	0
0	0	0	0	1
6	0	0	0	0
0	3	0	0	0
0	0	0	-4	0

(2, 5, 1), (3, 1, 6), (4, 2, 3), (5, 4, -4)

为了运算的方便,还可以根据题目的实际需要,将三元组线性表按照行号、列号等关键字进行排序,使其具有有序性,同时为了反映出矩阵的特点,可以在三元组存储的最前面增加一个元素 (n, m, k), 其中 n 表示矩阵的行, m 表示矩阵的列, k 表示非零元素的个数。

如上例的稀疏矩阵即为: (5, 5, 4), (2, 5, 1), (3, 1, 6), (4, 2, 3), (5, 4, -4)。

## 二、稀疏矩阵的链接存储

稀疏矩阵的链接存储就是对其相应的三元组线性表进行链接存储,比如按照行号或者列号来进行链接。

每个三元组结点的类型可定义为:

```
type matnode = record
    row, col : integer
    val : elementype;
    next: ^matnode
end;
```

其中 row, col, val 域分别存储三元组中的行号、列号和元素值, next 域存储指向下一个结点的指针,当然最后一个结点的 next 域为空。

比如把具有相同行号的三元组结点顺序链接成一个单链表,那么每行都有一个单链表,其中链接了该行中所有的有用元素。

这样每个单链表还需要定义一个指针向量,该向量中第 i 个分量用来存储第 i 个单链表的表头指针。该指针向量可以定义为:

```
type vectype = array [1..t] of ^matnode;
```

其中 t 表示单链表个数。

同样,根据题目实际情况,也可以把每个单链表内链接的元素按照一定的顺序排列,以方便运算。

## 三、稀疏矩阵的运算

这里将讨论稀疏矩阵的转置运算。

设一个稀疏矩阵 M, 三元组表示为数组 A, 它的转置矩阵为 N, 三元组表示为数组 B。数组 A 是以行号为第一关键字、列号为第二关键字从小到大排序的。

首先我们看到最朴素的转置算法。

此算法中,我们需要对数组 M 进行 n 次运算。具体地说,第一次扫描把第二列中其值等于 1

(即列号为 $i$ )所在的三元组(即对应 $N$ 中第一行非0元素所构成的三元组)按照从 $i$ 到下的顺序写入到数组 $B$ 中,第二次扫描把第二列中其值等于2所在的三元组(即对应 $N$ 中第二行非零所构成的三元组)按照从上到下的顺序写入到数组 $B$ 中,依此类推。此算法的复杂度是 $O(n * t)$ , $t$ 为非零元素个数。由此看出此算法的复杂度与和列数与非零元素的乘积个数成正比。

下面来介绍一种更优秀转置的算法。该算法需要对数组 $A$ 进行两次扫描。第一次扫描统计出 $M$ 中每一列(即对应 $N$ 中每一行)有用元素的个数,由此求出每一列的第一个有用元素在数组 $B$ 中的位置。第二次扫描把数组 $A$ 中的每一个三元组写入数组 $B$ 中确定的位置。

设 $col$ 表示 $M$ 中的列号(即对应 $N$ 中的行号), $num$ 和 $pot$ 均表示具有 $n$ ( $n$ 为 $M$ 中的列数即 $N$ 中的行数)各分量的向量, $num$ 向量的第 $col$ 个分量用来统计第 $col$ 列中的有用元素个数。 $pot$ 向量的第 $col$ 个分量用来指向第 $col$ 列的下一个有用元素在数组 $B$ 中的存储位置(即行号),显然 $pot$ 向量的第 $col$ 个分量的初始值(即第 $col$ 列的第一个有用元素在数组 $B$ 中的存储位置)应由下式计算:

$$\begin{cases} pot[1] = 1 \\ pot[col] = pot[col-1] + num[col-1] \quad (2 \leq col \leq n) \end{cases}$$

下面给出其算法描述

procedure fasttrans ( $A, B$ );

begin

(1)  $m := A[0, 1]; n := A[0, 2]; t := A[0, 3];$

(2)  $B[0, 1] := n; B[0, 2] := m; B[0, 3] := t;$

(3) if  $t = 0$  then return;

(4) for  $col := 1$  to  $n$  do

$num[col] := 0;$

(5) for  $i := 1$  to  $t$  do

$num[A[i, 2]] := num[A[i, 2]] + 1;$

(6)  $pot[1] := 1;$

for  $col := 2$  to  $n$  do

$pot[col] := pot[col-1] + num[col-1];$

(7) for  $i := 1$  to  $t$  do

(I)  $col := A[i, 2]; q := pot[col];$

(II)  $B[q, 1] := A[i, 2]; B[q, 2] := A[i, 1]; B[q, 3] := A[i, 3];$

(III)  $pot[col] := pot[col] + 1;$

end;

该算法的运行时间主要取决于第(4)步到第(7)步这四个并列的循环,故时间复杂度为 $O(n+t)$ ,显然已达到了理论下界,是一个高效的算法。

### 5.3 特殊矩阵的压缩存储

在实际应用中, 对于一个  $n \times m$  的矩阵, 往往是用一个二维数组保存下来, 这样即保存了所有的  $n \times m$  个元素。而有时候, 并不需要把所有的元素都保存下来, 或许只有一部分元素有用, 或许某一部分元素可以由另一部分元素推出, 这样我们就想到了对某些特殊矩阵进行压缩存储。

#### 一、对称矩阵的存储

有些矩阵是对称的, 那么我们只需要保存其中的一半, 另一半由这一半对称得到。比如下面是一个 5 个结点的无向图的边的矩阵:

0	2	3	5	4
2	0	10	2	1
3	10	0	7	6
5	2	7	0	4
4	1	6	4	0

这是一个关于主对角线 (左上到右下) 轴对称的矩阵, 我们只需保存其右上半部分, 左下半部分就可以很容易得到。也就是说我们保存了所有边  $\langle i, j \rangle$  ( $i < j$ ), 而对于  $i > j$  的边则不需要保存, 这样可以省下一半的空间。

#### 二、即时计算的矩阵

在用动态规划解题的时候, 往往都需要一个  $n \times m$  的矩阵来保存所有的状态。而有时候第  $i$  行的状态值和第  $i-1$  行有关, 而和  $i-2$  行以及更前面的状态没有关系, 如果目标是元素  $F_{[n,m]}$  的值, 也和前面状态无关的话就可以不保存  $i-2$  行以及更前面的状态的值。

那么可以用滚动数组来存储。即在计算第  $i$  行的状态时, 用数组  $A$  保存第  $i-1$  行状态, 数组  $B$  保存第  $i$  行状态, 而后随着  $i$  的增加, 数组  $A$ 、 $B$  进行滚动。如图 5-3 所示:

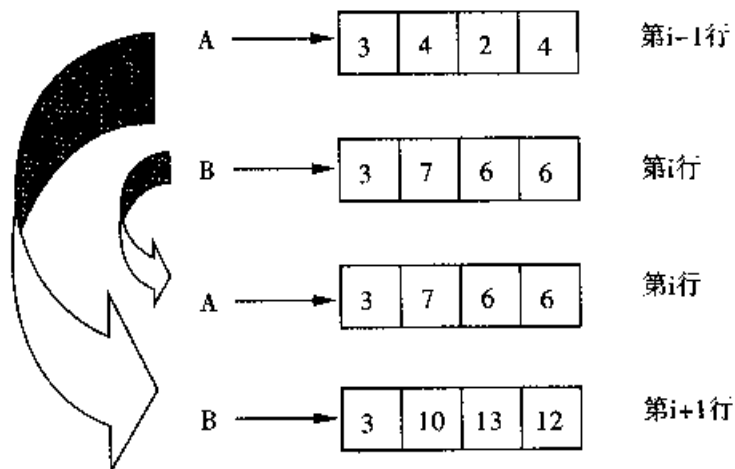


图 5-3 滚动数组存储示意图

### 三、稀疏矩阵的压缩存储

上一节中介绍过,稀疏矩阵即有用的元素远远少于无用的元素,这样可以用行号、列号、元素值这样的三元组来表示,可以避免浪费大量的空间来存储无用元素。实现时还可用线性表和链接存储两种方式,具体方法请参见上一节稀疏矩阵。

## 5.4 广义表

### 一、广义表的定义

广义表简称表,是线性表的推广。一个广义表是  $n$  个元素的一个序列,当  $n=0$  时称为空表。在一个非空的广义表中,其元素可以使某一确定类型的对象(这种元素被称做单元素),也可以是由单元素构成的表(这种元素可相对地被称做子表或表元素)。显然,广义表的定义是递归的,广义表是一种递归的数据结构。

设  $a_i$  为广义表的第  $i$  个元素,则广义表的一般表示为:

$$(a_1, a_2, \dots, a_i, \dots, a_n)$$

其中  $n$  表示广义表的长度,即广义表中所含元素的个数,  $n \geq 0$ 。

同线性表一样,也可以用一个标识符来命名一个广义表,如用  $LS$  命名什么的广义表,则为:

$$LS = (a_1, a_2, \dots, a_i, \dots, a_n)$$

在广义表的讨论中,为了将单元素同表元素区别开来,一般用小写字母表示单元素,用大写字母表示表,如:

$$A = ()$$

$$B = (e)$$

$$C = (a, (b, c, d))$$

$$D = (A, B, C) = (( ), (e), (b, c, d))$$

$$E = ((a, (a, b), ((a, b), c)))$$

其中  $A$  是一个空表,其长度为 0;  $B$  是只含一个单元素  $e$  的表,其长度为 1;  $C$  中有两个元素,一个是单元素  $a$ ,另一个是表元素  $(b, c, d)$ ,  $C$  的长度为 2;  $D$  中有三个元素,其中每个元素又都是一个表,  $D$  的长度为 3;  $E$  中只含有一个元素,该元素是一个表。

若把每个表的名字(若有的话)都写在其前面,则上面五个广义表可相应地表示为:

$$A ()$$

$$B (e)$$

$$C (a, (b, c, d))$$

$$D (A, B, C) = (( ), (e), (b, c, d))$$

$$E ((a, (a, b), ((a, b), c)))$$

### 二、广义表的存储结构

广义表表示一种递归的数据结构,因此很难为每个广义表分配固定大小的存储空间,所以其存储结构最好采用动态链接结构。

在一个广义表中,其数据元素有单元素和子表之分,所以在对应的存储结构中,其存储结点

也有单元素结点和子表结点之分。对于单元素结点,应包括值域和指向其后继结点的指针域;对于子表结点,应包括指向子表中第一个结点的表头指针域和指向其后继结点的指针域。为了把广义表中的单元素结点和子表结点区别开来,还必须在每个结点中增设一个标志域,让标志域取两种不同的值,从而代表两种不同的结点。

根据分析广义表中的结点类型可定义为:

```
type genenode = record
    tag : 0..1;
    next : ^genenode;
    case tag of
        0 : (data : elemtype);
        1 : (sublist : ^genenode)
    end;
```

其中 tag 作为标志域,取 0 表示单元素结点,取 1 表示子表结点;next 作为指向其后继结点的指针域,通过它把表中的所有结点一次链接起来;data 作为单元素的值域;sublist 作为指向本子表中第一个结点的表头指针域,通过它实现向子表的链接,亦即实现广义表的递归结构。

### 三、广义表的运算

广义表的运算主要有求广义表的长度和深度,向广义表插入元素和从广义表中查找或删除元素,建立广义表的存储结构,输出广义表等。由于广义表是一种递归的数据结构,所以对广义表的运算一般采用递归的算法。下面讨论广义表中求深度的算法。

广义表深度的递归定义是它等于所有子表中表的最大深度加 1。设 dep 表示任意子表的深度, max 表示所有子表中表的最大深度, depth 表示广义表的深度,则有:

$$\text{depth} = \max + 1$$

若一个表中不包含任何子表时,其深度为 1,所以 max 的初值为 0。

设 ls 是具有 ^genenode 类型的一个值参,开始时指向一个广义表的第一个结点,则求一个广义表的深度的递归算法如下:

```
function depth (ls) : integer;
begin
    max := 0;
    while ls <> nil do
        if ls^.tag = 1 then
            dep := depth (ls^.sublist);
            if dep > max then max := dep;
        ls := ls^.next;
        depth := max + 1;
    end;
```



## 5.5 小结

这一章中,介绍了多维数组,然后介绍稀疏矩阵这一种较特殊也较为常见的矩阵,对其两种存储方式以及转置运算都作了较为详细的介绍,介绍了特殊矩阵的压缩,最后学习了广义表的定义、存储及运算。

### 习题五

#### 一、选择题

1. 用行优先的方式存储一个  $100 \times 100$  的矩阵,假设存储每个元素需要一个字节的空间。如果  $a[1, 1]$  的存储地址是  $p$ ,那么  $a[1, 2]$  的存储地址应该是( )。

- A.  $p+1$                       B.  $p$                       C.  $p-1$                       D.  $p+2$

2. 用列优先的方式存储一个  $100 \times 100$  的矩阵,假设存储每个元素需要一个字节的空间。如果  $a[1, 1]$  的存储地址是  $p$ ,那么  $a[55, 55]$  的存储地址应该是( )。

- A.  $p+1$                       B.  $p+5455$                       C.  $p+5454$                       D.  $p+5555$

3. 对于下图中  $2 \times 3$  的矩阵,其转置矩阵为( )。

1	2
2	1
0	0

A.

1	2
2	1
0	0

B.

0	2	1
0	1	2

C.

1	2	0
2	1	0

D.

2	1	0
1	2	0

4. 对于  $5 \times 6$  的稀疏矩阵的三元组表示法  $(5, 6, 5)$ ,  $(1, 2, 4)$ ,  $(2, 5, 1)$ ,  $(3, 3, 6)$ ,  $(4, 2, 2)$ ,  $(4, 5, 1)$ , 对其进行转置后的三元组表示法为( )。

- A.  $(6, 5, 5)$ ,  $(2, 1, 4)$ ,  $(5, 2, 1)$ ,  $(3, 3, 6)$ ,  $(2, 4, 2)$ ,  $(5, 4, 1)$   
 B.  $(6, 5, 5)$ ,  $(2, 1, 4)$ ,  $(2, 4, 1)$ ,  $(3, 3, 6)$ ,  $(5, 2, 2)$ ,  $(5, 4, 1)$   
 C.  $(6, 5, 5)$ ,  $(2, 1, 4)$ ,  $(2, 4, 2)$ ,  $(3, 3, 6)$ ,  $(5, 2, 1)$ ,  $(5, 4, 1)$   
 D.  $(6, 5, 5)$ ,  $(2, 4, 2)$ ,  $(5, 2, 1)$ ,  $(2, 1, 4)$ ,  $(3, 3, 6)$ ,  $(5, 4, 1)$

5. 广义表  $A()$  的深度是( )。

- A. 0                      B. 1                      C. 2                      D. 3

6. 广义表  $A(B(c, d, e), C(D(E())))$  的深度是( )。



A. 1

B. 2

C. 3

D. 4

## 二、上机编程题

1. 求一个五阶方阵中这样的元素的位置：它在行上是最小的，在列上也是最小；如果没有请输出“NO FIND!”。

2. 给定一个  $N$  个点  $M$  条边的无向图，可以定义其  $N * M$  的关联矩阵  $A_{ij} = 1$  当且仅当第  $i$  个顶点是第  $j$  条边的一个端点，否则  $A_{ij} = 0$ 。同时定义  $A^T$  是一个  $M * N$  的矩阵，满足  $A_{ij}^T = A_{ji}$ ，也就是  $A$  的转置矩阵。求  $A^T A$  中的全部元素的和。其中  $N \leq 100000$ ， $M \leq 1000000$ 。

3. 你想编写一个操作系统 WinStack，这个操作系统应该具有世界上最强的栈模拟功能：它能维护 1000 个堆栈，并且可以在 2s 内进行 1000000 万次操作。操作有两种格式：

PUSH Num key：表示把元素 key 插入编号为 Num 的栈中，其中  $key \leq 2^{30}$ ；

POP Num：表示将编号为 Num 的栈的栈顶元素出栈，输出这个值。

每个栈的大小都应该是不确定的，但是你想要尽可能地节约空间，因此你只能使用 10MB 的内存空间，并且只能向指定文件进行输出。

4. 所谓“幻方”，是一个行列数为奇数的方阵，把  $1 \sim n^2$  这  $n^2$  个不同的数放入方阵中，使方阵的每行、每列和每个对角线上的元素的和全部相等。下面给出一个阶幻方和一种排列方法：2 这  $n^2$  个不同的数放入方阵中，使方阵的每行、每列和每个对角线上的元素的和全部相等。下面给出一个阶幻方和一种排列方法：2 这  $n^2$  个不同的数放入方阵中，使方阵的每行、每列和每个对角线上的元素的和全部相等。下面给出一个阶幻方和一种排列方法：

(1) 先把放在第一行的中间位置。

(2) 下一个放在上一个数的右上方。

(3) 若右上方已超出方阵的第一行，则下一个数放在下一列的最后一行上。

(4) 若右上方已超出方阵的最后一列，则一个数放在上一行的第一列上。

(5) 若右上方已经有数，或右上方已超出方阵的第一行最后一列，则下一个数放在上一个数的正下方。

编程序，对输入小于 15 的  $n$ ，打印出相应的幻方。



## 6 树

### 6.1 树的概念

#### 一、树的概念

树 (tree) 是树型结构的简称。它是一种重要的非线性数据结构。树——或者是一棵空树，即不含结点的树，或者是一棵非空树，即至少含有一个结点的树。在一棵非空树中，它有且仅有一个称作根 (root) 的结点，其余的结点可分为  $m$  棵 ( $m \geq 0$ ) 互不相交的子树 (即称作根的子树)，每棵子树 (subtree) 又同样是一棵树。显然，树的定义是递归的，树是一种递归的数据结构。树的递归定义，将为以后实现树的各种运算提供方便。

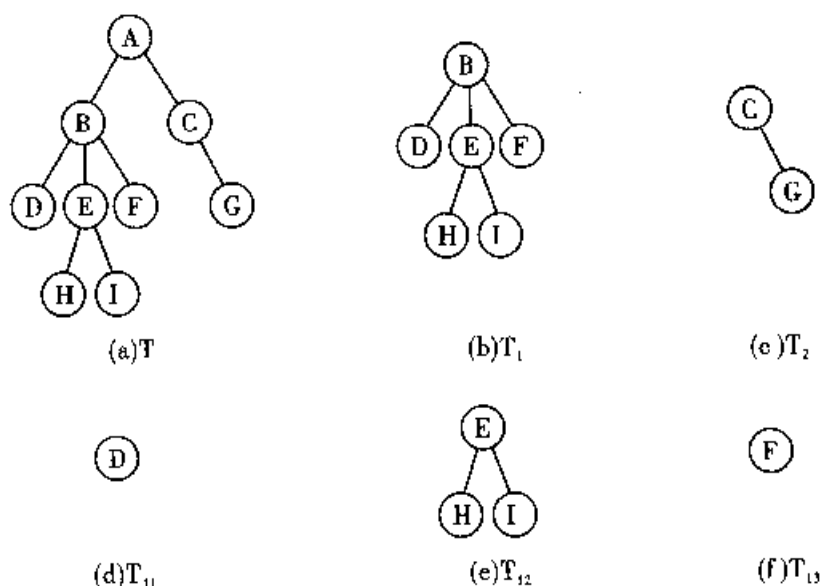


图 6-1 树的结构

图 6-1 (a) 就是一棵树  $T$ ，它由根结点  $A$  和两棵子树  $T_1$  和  $T_2$  (分别对应图 6-1 (b) 和 (c)) 所组成； $T_1$  又由它的根结点  $B$  和三棵子树  $T_{11}$ 、 $T_{12}$  和  $T_{13}$  (分别对应图 6-1 (d)、(e)、(f)) 所组成； $T_{11}$  和  $T_{13}$  只含有根结点，不含有子树 (或者说子树为空树)，即不可再分； $T_{12}$  又由它的根结点  $E$  和两棵只含有根结点的子树所组成，每棵子树的根结点分别为  $H$  和  $I$ ； $T_2$  由它的根结点  $C$  和一棵子树所组成，该子树也只含有一个根结点  $G$ ，不可再分。

在一棵树中，每个结点被定义为它的子结点的前驱，而它的每个子结点又是它的后继。由此，可以用二元组定义一棵树：

$$\text{Tree} = (K, R)$$



$K = \{k_i | 1 \leq i \leq n, n \geq 0, n \text{ 为树中的结点数}, k_i \in \text{elemtype}\}$

$R = \{r\}$

当  $n > 0$  (即树为非空树) 时, 关系  $r$  应满足下列条件:

- (1) 有且仅有一个结点没有前驱, 该结点被称为树的根;
- (2) 除树根结点外, 其余每个结点有且仅有一个前驱结点;
- (3) 包括树根结点在内的每个结点, 可以有任意多个 (含 0 个) 后继结点。

对于图 6-1 (a) 所示的树  $T$ , 若采用二元组表示, 则结点的集合  $K$  和  $K$  上的二元关系  $r$  分别为:

$K = \{A, B, C, D, E, F, G, H, I\}$

$r = \{ \langle A, B \rangle, \langle A, C \rangle, \langle B, D \rangle, \langle B, E \rangle, \langle B, F \rangle, \langle C, G \rangle, \langle E, H \rangle, \langle E, I \rangle \}$

其中  $A$  结点无前驱结点, 被称为树的根结点; 其余每个结点有且仅有一个前驱结点; 在所有结点中,  $B$  结点有三个后继结点,  $A$  结点和  $E$  结点分为有两个后继结点,  $C$  结点有一个后继结点, 其余结点均没有后继结点。

在日常生活中, 树结构广泛存在。

**例题 6-1** 可把一个家族看作为一棵树, 树中的结点为家族成员的姓名及相关信息, 树中的关系为父子关系, 即父亲是儿子的前驱, 儿子是父亲的后继。图 6-2 就是一棵家族树, 王新贵有两个儿子王万和和王万田, 王万和又有三个儿子王家利、王家中和王家国。

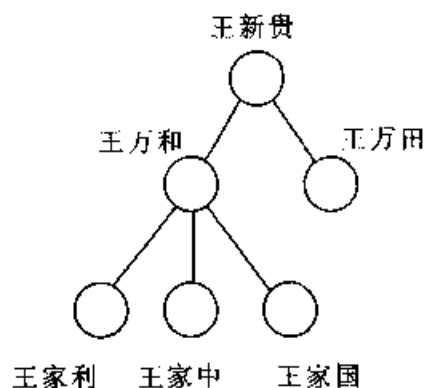


图 6-2 家族树

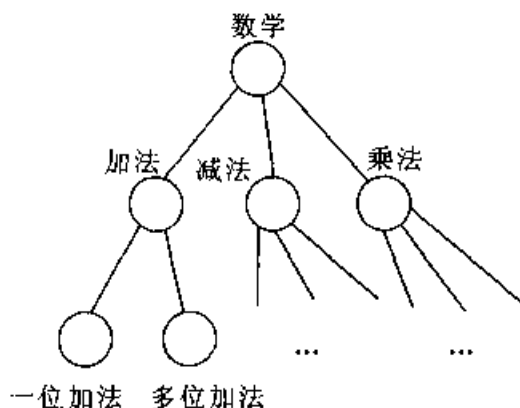


图 6-3 树状结构的书

**例题 6-2** 可把一个国家或一个地区的各级行政区划看作为一棵树, 树中的结点为行政区的名称及相关信息, 树中的关系为上下级关系。如一个城市包含有若干个区, 每个区又包含有若干个街道, 每个街道又包含有若干个居委会等。

**例题 6-3** 可把一本书的结构看作为一棵树, 树中的结点为书、章、节的名称及相关信息, 树中的关系为包含关系。图 6-3 是一本书的结构, 根结点为书的名称数学, 它包含三章, 每章名称分别为加法、减法和乘法, 加法一章又包含两节, 分别为一位加法和多位加法, 减法和乘法也分别包含若干节。

**例题 6-4** 可把一个算术表达式表示成一棵树, 运算符作为根结点, 它的前后两个运算对象分别作为根的左、右两棵子树。如把算术表达式

$$a * b + (c - d / e) * f$$

表示成树，则如图 6-4 所示。

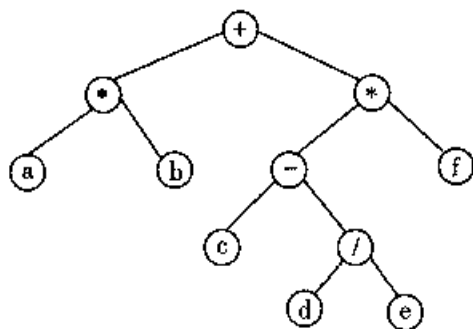


图 6-4 表达式树

## 二、树的表示

树的表示方法有多种。图 6-1 至图 6-4 中的树形表示法是其中的一种，也是最常用的一种。在这种表示法中，结点之间的关系是通过连线表示的，虽然每条连线上都不带有箭头（即方向），但它并不是无向的，而是有向的，其方向隐含为从上向下，即连线的上方结点是下方结点的前驱，下方结点是上方结点的后继。树的另一种表示法是三元组表示法。除这两种之外，通常还有三种：一是集合图表示，每棵树对应一个圆形，圆内包含根结点和子树，图 6-1 的树 T 对应的集合图表示如图 6-5 (a) 所示；二是凹入表表示，每棵树的根对应着一个条形，子树的根对应着一个较短的条形，且树根在上，子树的根在下，树 T 的凹入表表示如图 6-5 (b) 所示；三是广义表表示，每棵树的根作为由子树构成的表的名字而放在表的左边，树 T 的广义表表示如图 6-5 (c) 所示。

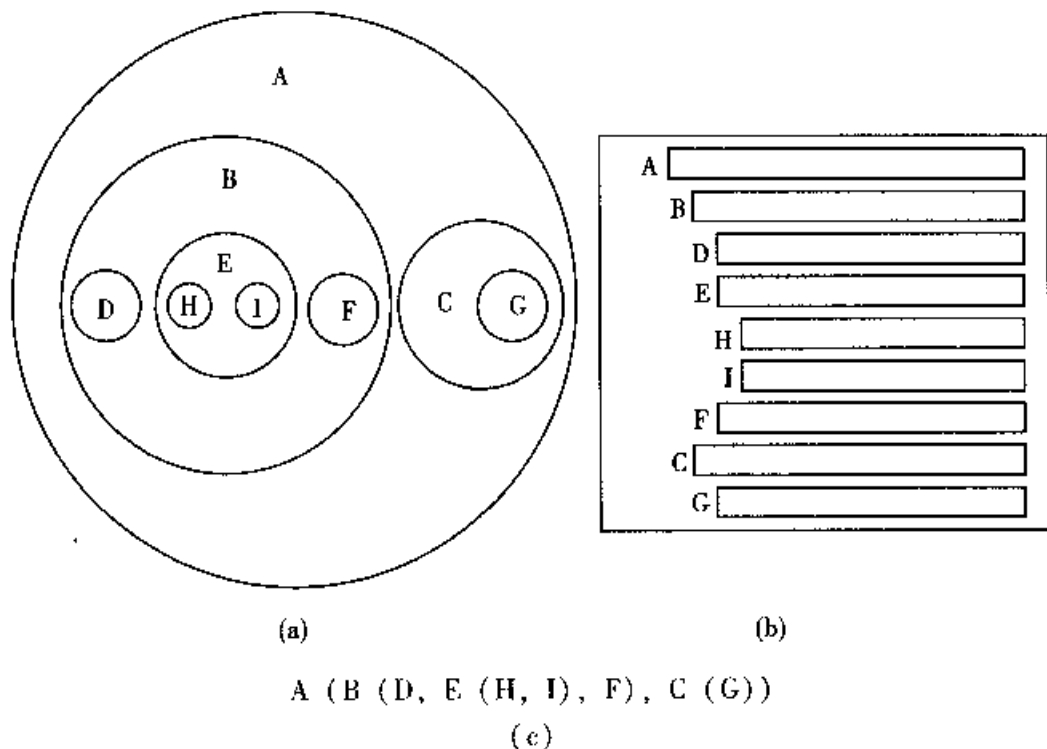


图 6-5 树的其他表示

### 三、树的基本术语

#### 1. 结点的度和树的度

每个结点具有的子树数或者说后继结点数被定义为该结点的度 (degree)。所有结点的度的最大值被定义为该树的度。如在图 6-1 的树 T 中, B 结点的度为 3, A、E 结点的度都为 2, C 结点的度为 1, 其余结点的度均为 0。因结点的最大度为 3, 所以树 T 的度为 3。

#### 2. 分支结点和叶子结点

度大于 0 的结点称作分支结点或非终端结点, 度等于 0 的结点称作叶子结点或终端结点。在分支结点中, 又把度为 1 的结点叫做单分支结点, 度为 2 的结点叫做双分支结点, 其余依此类推。如在图 6-1 的树 T 中, A、B、C、E 都是分支结点, D、H、I、F、G 都是叶子结点; 在分支结点中, C 为单分支结点, A、E 分别为双分支结点, B 为三分支结点。

#### 3. 孩子结点、双亲结点和兄弟结点

每个结点的子树的根, 或者说每个结点的后继, 被习惯地称作该结点的孩子 (child) 或儿子, 相应地, 该结点被称作孩子结点的双亲 (parent) 或父亲。具有同一双亲的孩子互称兄弟 (brothers)。每个结点的所有子树中的结点被称作该结点的子孙。每个结点的祖先则被定义为从树根结点到达该结点的路径上经过的所有结点。如在图 6-1 的树 T 中, B 结点的孩子为 D、E、F 结点, 双亲为 A 结点, D、E、F 互为兄弟, B 结点的子孙为 D、E、H、I、F 结点, I 结点的祖先为 A、B、E 结点, 对于树 T 中的其他结点亦可进行同样的分析。

由孩子结点和双亲结点的定义可知: 在一棵树中, 根结点没有双亲结点, 叶子结点没有后继结点。如在图 6-1 的树 T 中, A 结点没有双亲结点, D、H、I、F、G 结点没有孩子结点。

#### 4. 结点的层数和树的深度

树既是一种递归结构, 也是一种层次结构, 树中的每个结点都处在一定的层数上。结点的层数 (level) 从树根开始定义, 根结点为第一层, 它的孩子结点为第二层, 依此类推。树中结点的最大层数称为树的深度 (depth) 或高度 (height)。如在图 6-1 的树 T 中, A 结点处于第一层, B、C 结点处于第二层, D、E、F、G 结点处于第三层, H、I 结点处于第四层。H、I 结点所处的第四层为树 T 中结点的最大层数, 所以树 T 的深度为 4。

#### 5. 有序树和无序树

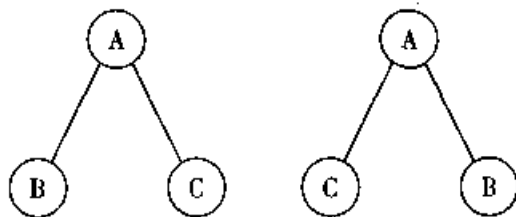


图 6-6 两棵不同的有序树

若树中各结点的子树是按照一定的次序从左向右安排的, 则称之为有序树, 否则称之为无序树。如对于图 6-6 中的两棵树, 若看作为无序树, 则是相同的; 若看作为有序树, 则不同, 因为根结点 A 的两棵子树的次序不同。又如, 对于一棵反映父子关系的家族树, 兄弟结点之间是按照排行大小有序的, 所以它是一棵有序树。再如, 对于一个机关或单位的机构设置树, 若各层机构是按照一定的次序排列的, 则为一棵有序树, 否则为一棵无序树。因为任何无序树都可以当作任一次序的有序树来处理, 所以以后若不特别指明, 均认为树是有序的。

## 6. 森林

森林是  $m$  ( $m \geq 0$ ) 棵互不相交的树的集合。例如, 对于树中每个分支结点来说, 其子树的集合就是森林。在图 6-1 的树  $T$  中, 由  $A$  结点的子树所构成的森林为  $\{T_1, T_2\}$ , 由  $B$  结点的子树所构成的森林为  $\{T_{11}, T_{12}, T_{13}\}$ , 等等。

## 6.2 二叉树

### 一、二叉树的定义

二叉树 (binary tree) 是指树的度不超过 2 的有序树。它是一种最简单、而且最重要的树。二叉树的递归定义为: 二叉树或者是一棵空树, 或者是一棵由一个根结点和两棵互不相交的分别称作根的左子树和右子树所组成的非空树, 左子树和右子树又同样都是二叉树。

图 6-7 (a) 就是一棵二叉树  $BT$ , 它由根结点  $A$  和左子树  $BT_1$  (对应图 6-7 (b))、右子树  $BT_2$  (对应图 6-7 (c)) 所组成,  $BT_1$  又由根结点  $B$  和左子树  $BT_{11}$  (只含有根结点  $D$ )、右子树  $BT_{12}$  (为空树) 所组成; 对于  $BT_2$  树也可进行类似的分析。

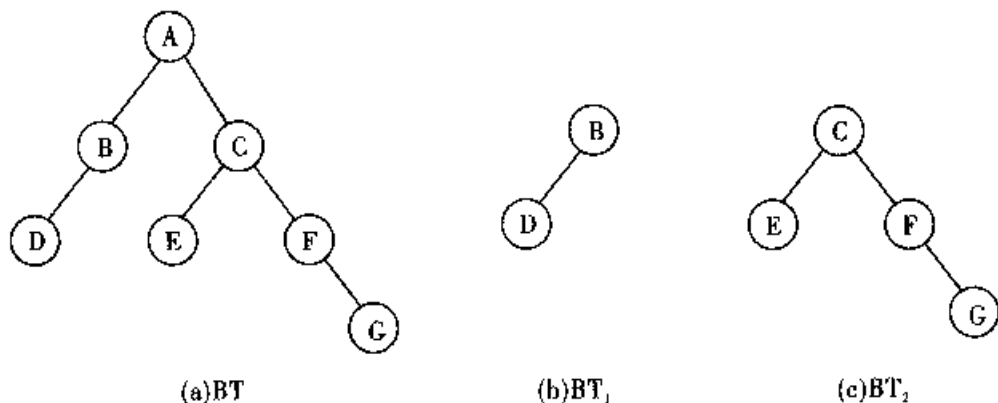


图 6-7 二叉树示例

在二叉树中, 每个结点的左子树的根结点被称之为左孩子 (left child), 右子树的根结点被称之为右孩子 (right child)。如在图 6-7 的  $BT$  二叉树中,  $A$  结点的左孩子为  $B$  结点, 右孩子为  $C$  结点;  $B$  结点的左孩子为  $D$  结点, 右孩子为空, 或者说没有右孩子;  $C$  结点的左孩子为  $E$  结点, 右孩子为  $F$  结点;  $F$  结点没有左孩子, 右孩子为  $G$  结点。

### 二、二叉树的性质

二叉树具有下列重要性质:

性质 1: 二叉树上第  $i$  层上至多有  $2^{i-1}$  个结点 ( $i \geq 1$ )。

下面分为  $i=1$  和  $i>1$  两种情况讨论:

当  $i=1$  时,  $2^{i-1} = 2^0 = 1$ , 因为二叉树的第一层上只有一个结点 (即根结点), 故命题成立。

当  $i>1$  时, 假定第  $i-1$  层上的结点数至多有  $2^{(i-1)-1} = 2^{i-2}$  个, 根据二叉树的定义, 每个结点至多有两个孩子, 所以第  $i$  层上的结点数至多为第  $i-1$  层上结点数的 2 倍, 即  $2 \times 2^{i-2} = 2^{i-1}$ , 命题成立。

性质 2: 深度为  $h$  的二叉树至多有  $2^h - 1$  个结点。

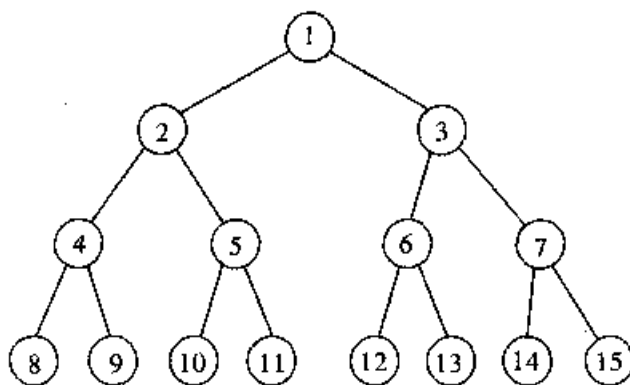
显然,当深度为  $h$  的二叉树上每一层都达到最多的结点数时,它们的和才能最大,即整个二叉树才具有最多结点数:

$$\sum_{i=1}^h 2^{i-1} = 2^0 + 2^1 + 2^2 + \dots + 2^{h-1} = 2^h - 1$$

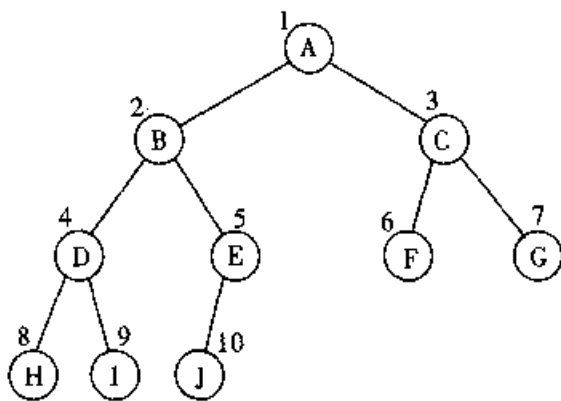
故命题成立。

在一棵二叉树,如果它的第  $i$  层的结点数达到了  $2^{i-1}$  个,则称这棵树的第  $i$  层是满的。当二叉树中的每层都是满的时,就称此二叉树为满二叉树。由性质 2 的证明可知,深度为  $h$  的满二叉树的结点数为  $2^h - 1$  个,而且,深度为  $h$  的二叉树中只有满二叉树的结点数能达到  $2^h - 1$ 。

图 6-8 (a) 中是一棵深度为 4 的满二叉树,它有 15 个结点。图中的结点是用数值来标号的,通常,满二叉树的标号是按层数从小到大,同一层按从左到右的顺序标号。



(a)



(b)

图 6-8 满二叉树和完全二叉树

在一棵二叉树中,如果除最后一层外,其他层都是满的,且最后一层或是满的,或是所有结点都集中在左边,则这棵二叉树称为完全二叉树。显然,满二叉树是特殊的完全二叉树。如图 6-8 (b) 和图 6-9 都是一棵完全二叉树。

若对完全二叉树按满二叉树的方法标号,有:

完全二叉树的标号性质 1: 在  $n$  个结点的完全二叉树中,对于标号为  $i$  的结点,若  $i \leq \lfloor n/2 \rfloor$ , 则结点  $i$  为分支结点,否则为叶子结点。

完全二叉树的标号性质 2: 在  $n$  个结点的完全二叉树中,若  $n$  为奇数,则所有的分支结点都有左孩子和右孩子;若  $n$  为偶数,则编号为  $n/2$  的结点只有左孩子,没有右孩子,其他分支结点既

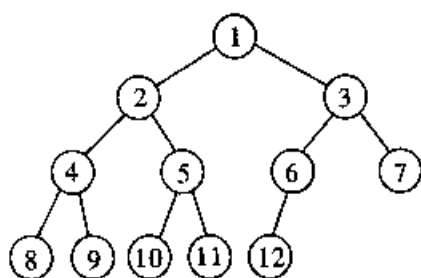


图 6-9 一棵完全二叉树

有左孩子又有右孩子。

完全二叉树的标号性质 3: 若标号为  $i$  的结点有左孩子, 则它的左孩子为  $2i$ ; 若标号为  $i$  的结点有右孩子, 则它的右孩子为  $2i+1$ 。

完全二叉树的标号性质 4: 若标号为  $i$  的结点有双亲结点, 则  $i > 1$  且它的双亲结点标号为  $\lfloor i/2 \rfloor$ 。

完全二叉树的深度性质 具有  $n$  ( $n > 0$ ) 的结点的完全二叉树的深度为  $\lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil$ 。

证: 因为深度为  $h$  的二叉树的结点个数最多为  $2^h - 1$  个, 又深度为  $h$  的完全二叉树的结点个数最少为  $2^{h-1}$  个, 设  $n$  个结点的完全二叉树深度为  $h$ , 则:

$$2^{h-1} \leq n < 2^h$$

取对数, 得:  $h-1 \leq \log_2 n < h$

因为  $h$  是整数, 不难得到:  $h = \lfloor \log_2 n \rfloor + 1$

因为  $\lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil$ , 所以  $h = \lceil \log_2 (n+1) \rceil$  亦成立。

在一棵二叉树中, 若除最后一层外, 其余层都是满的, 则称此树为理想平衡树。显然, 完全二叉树是特殊的理想平衡树。图 6-10 (a) 是一棵理想平衡树, 而图 6-10 (b) 不是, 因为它的倒数第二层没有满。

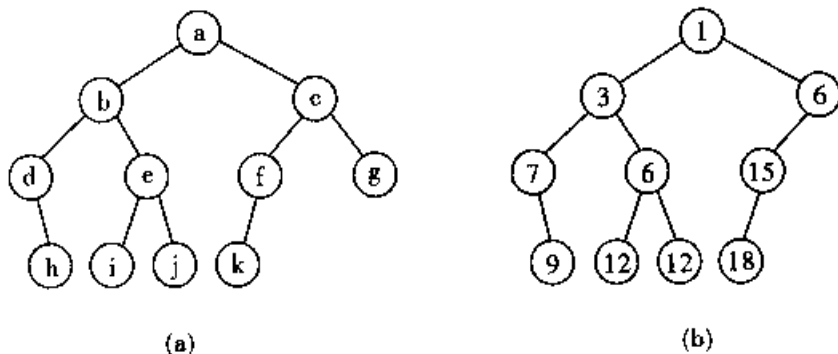


图 6-10 理想平衡树与非理想平衡树

### 三、二叉树的存储结构

二叉树具有线性存储和链接存储两种存储结构。

#### 1. 线性存储

顺序存储一棵二叉树时, 首先要将二叉树按照完全二叉树中对应的位置进行标号, 然后, 以每个结点的标号为下标, 将对应的值存储到一个一维数组中。由完全二叉树标号的性质可知, 根

结点的标号为1, 如果标号为 $i$ 的结点有左孩子结点, 那么它的左孩子结点标号为 $2i$ ; 如果标号为 $i$ 的结点有右孩子, 那么它的右孩子结点标号为 $2i+1$ 。这样, 就可以按层次从上到下的顺序给每一层标号。如图6-11就是一个标号的例子, 结点里是结点的值, 边上的值是结点的编号。

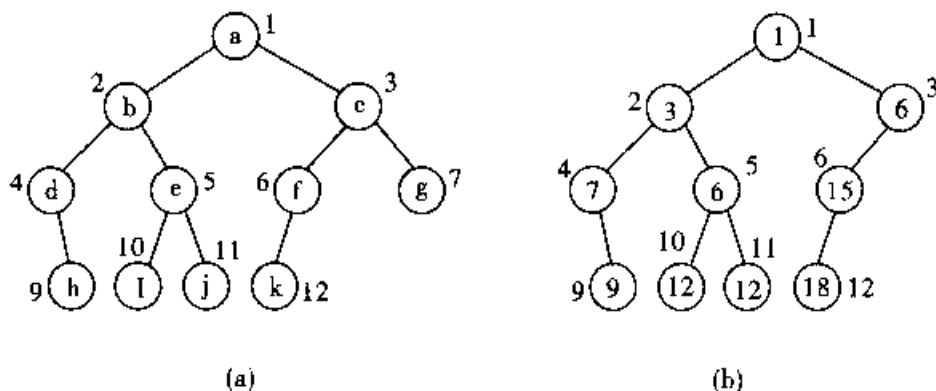


图6-11 二叉树的结点编号

若将图6-11的两棵二叉树分别存放在数组TreeA和TreeB中, 则两个数组如下:

数组下标	1	2	3	4	5	6	7	8	9	10	11	12
TreeA	a	b	c	d	e	f	g		h	i	j	k
TreeB	1	3	6	7	6	15			9	12	12	18

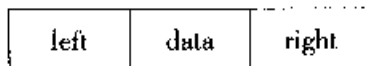
注意: 在数组中可能有一些空的元素, 表示这个结点为空。

在线性存储的二叉树中, 各结点之间的关系可以通过下标表示出来, 如下标为 $i$ 的结点的父结点是下标为 $\lfloor i/2 \rfloor$ 的结点, 左孩子是下标为 $2i$ 的结点, 而右孩子是下标为 $2i+1$ 的结点。

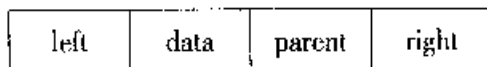
当一棵树是完全二叉树时, 用顺序存储是一种非常好的选择, 它实现简单、不易出错, 而且能充分利用空间。但是, 对于一般的二叉树, 特别是单分支结点比较多的, 如果用顺序存储, 必然造成空间的极大浪费。

## 2. 链接存储

二叉树有另一种存储方式, 那就是链接存储。当一棵二叉树用链接存储时, 每个结点都要设置一个域: 值域 (data)、左指针 (left) 域和右指针 (right) 域。其中, 值域用于存储结点的值, 左指针域用于存储一个指向它的左孩子的指针, 右指针域用于存储一个指向它的右孩子的指针。通常一个结点表示如下:



如果在二叉树中希望能及时找到一个结点的父结点, 那么通常在存储结构中加入一个 parent 指针指向它的父结点, 即:



如图6-12中, (a) 是一棵二叉树, (b) 是用第一种链接存储, (c) 是用第二种链接存储。

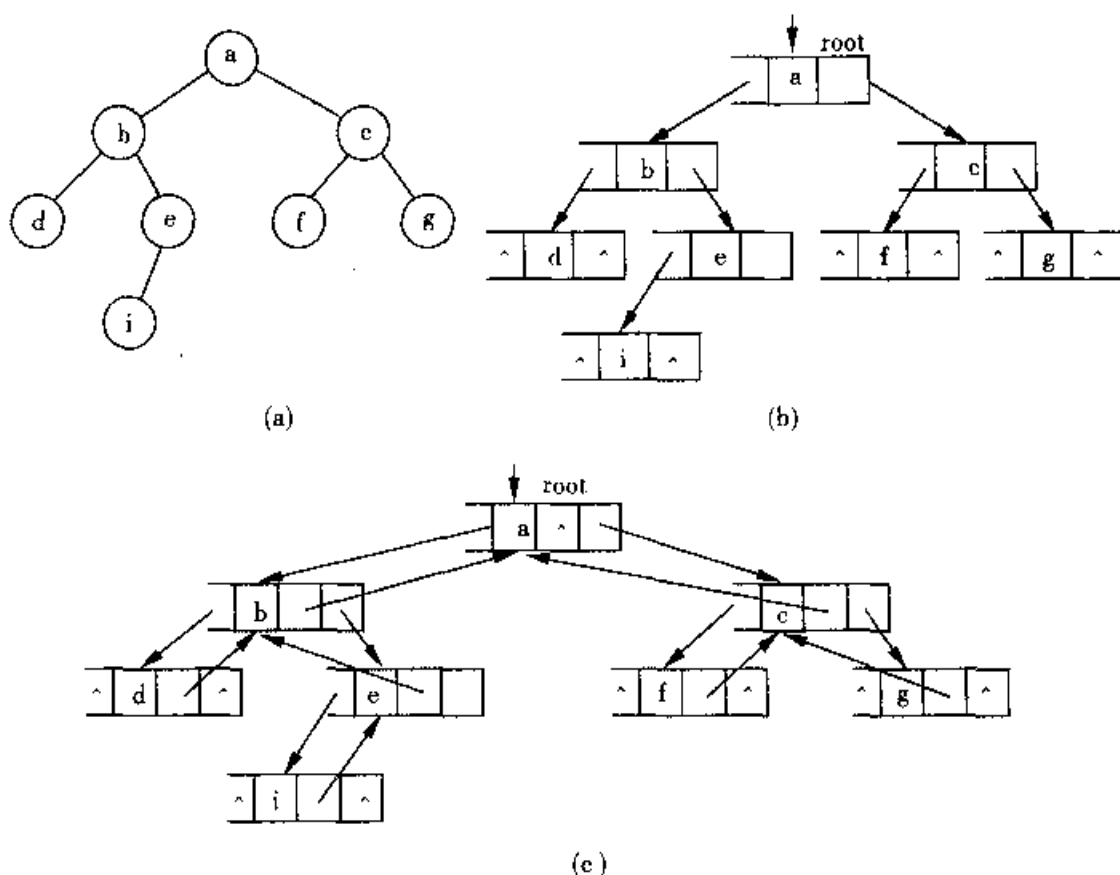


图 6-12 二叉树的结点编号

和单链表一样，二叉树的链接表既可以由静态结点链接而成，也可以由动态结点链接而成。它们的结点定义分别如下：

二叉树的静态结点定义：

Type

TNode = record

data : elementType;

left, right : integer;

end;

使用静态结点必须使用一个数组来存储。在结点的定义中，left 和 right 为左、右孩子结点在数组中所在的单元的下标，所以用整型。其数组类型的定义如下：

type

TTree = array [1..Size] of TNode;

二叉树的动态结点定义：

Type

PNode = ↑ TNode;

TNode = record

data : elementType;

left, right : PNode;





end;

由于是链接存储,在一个二叉树中,结点的存放顺序可以是任意的,以静态链接为例,图6-12的树可以存储为:

下标	1	2	3	4	*5	6	7	8	...	Size
data	d	b	i	e	a	F	c	g	...	
left	0	1	0	3	2	0	6	0	...	
right	0	4	0	0	8	0	8	0	...	

#### 四、由广义表生成二叉树

二叉树的生成根据输入的不同而不同,这里,我们介绍如何将一个广义表表示的二叉树表示成一棵用动态链接表存储的二叉树。

首先,有必要对广义表的表示规范一下,有助于我们后面的处理:每棵树的根结点放在最前面;若它有子树,则在它后面加一对圆括号,子树按从左子树到右子树的顺序写入,并在左右子树之间放一个逗号(,)隔开;若某个结点只有左子树而没有右子树,则省略逗号和右子树;若某个结点只有右子树而没有左子树,则左子树省略,但不能省略逗号;在整个树的后面加入一个@作为结束符。

算法的基本思想是:依次输入广义表中的每个字符,若遇到字母(假定以字母为结点的值),则表明是结点的值,就为它建立一个新的结点,并把它作为根(第一个字母时)或作为孩子结点连接到它的父结点上;若遇到的是左括号,则表明子树表开始,应首先将它前面字母的结点的指针(即此子树的根)进栈,以便作为以后结点的父结点指针引用,并记下下一个要插入的结点应为其父结点的左孩子( $k=1$ );若遇到右括号,则表明子树表结束,应退栈;若遇到逗号,则表明以左子树处理完毕,应处理其父结点的右孩子( $k=2$ )。这样处理每一个字符,直到处理到@说明输入结束。

其算法的伪代码为:

procedure buildtree;

begin

    输入一个字符 ch

    while (ch 不为@) do

        begin

            case ch of

                字母:新建结点,值域的值ch,左右子树指针为空;

                如果这是第一个结点,那么它是根结点,否则若 $k=1$ ,则将当前结点作为栈顶结点的左子树,否则( $k=2$ ),则将当前结点作为栈顶结点的右子树。

                ‘(’:将当前的结点插入栈顶;设下一次处理的为右子树( $k \leftarrow 1$ )

                ‘)’:栈顶元素出栈。

                ‘,’:设下一次处理的为右子树( $k \leftarrow 2$ )。

        end;



输入下一个字符 ch

end;

end;

## 6.3 二叉树的运算

二叉树的运算主要包括二叉树的遍历、二叉树的输出、求二叉树的深度等。

为了讨论算法的方便, 以下我们的操作都建立在动态链接表上, Node 是一个具有 PNode 类型的参数, 并且最开始是指向一棵二叉树的根结点。

### 一、二叉树的遍历

二叉树的遍历是指按照一定的顺序访问二叉树的每一个结点, 并且每个结点只访问一次。它是二叉树中一个非常重要的内容。

由二叉树的递归定义, 一棵非空二叉树由根结点、左子树和右子树组成。若用 D、L 和 R 分别表示根和左、右子树, 则二叉树的遍历一共有六种: DLR、LDR、LRD、DRL、RDL、RLD, 其中前三种都是先遍历左子树、后遍历右子树, 而后三种相反。由于前三种出现得比较普遍, 而后三种很少出现, 且前三种和后三种具有对称性, 这里只讨论前三种遍历。

在遍历 DLR 中, 因根先于左、右子树遍历, 故称为前序遍历 (preorder) 或先根遍历; 在遍历 LDR 中, 因根在左、右子树遍历之间, 故称为中序遍历 (inorder) 或中根遍历; 在遍历 LRD 中, 因根后于左、右子树遍历, 故称为后序遍历 (postorder) 或后根遍历。显然, 遍历左右子树仍然是遍历二叉树的问题, 所以很容易给出这三种遍历的递归算法。

#### 1. 前序遍历算法

```
procedure preorder (Node);
begin
    if Node <> nil then
        PROCESS (Node);
        preorder (Node^.left);
        preorder (Node^.right);
    end;
```

#### 2. 中序遍历算法

```
procedure inorder (Node);
begin
    if Node <> nil then
        inorder (Node^.left);
        PROCESS (Node);
        inorder (Node^.right);
    end;
```

#### 3. 后序遍历算法

```

procedure postorder (Node);
begin
  if Node < > nil then
    postorder (Node^.left);
    postorder (Node^.right);
    PROCESS (Node);
end;

```

其中 PROCESS (Node) 表示对 Node 进行的处理, 这个处理是什么要根据具体情况而定。

对图 6-13 中的树, 分别执行前序、中序和后序, 可以得到三个遍历的序列: 前序序列 (A, B, D, E, H, C, F, G), 中序序列 (D, B, H, E, A, F, C, G), 后序序列 (D, H, E, B, F, G, C, A)。

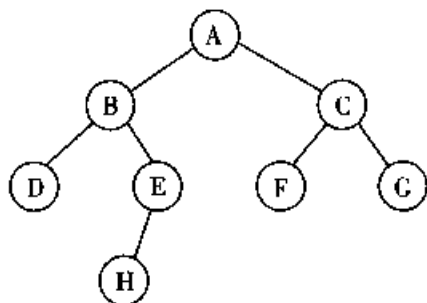


图 6-13 二叉树示意图

## 二、输出二叉树

输出二叉树就是将二叉树以某种表示形式打印出来。下面, 我们介绍如何将二叉树以广义表的形式输出。

以广义表的形式输出时, 首先应输出树的根结点; 然后加一个左括号; 如果左子树非空则输出左子树; 如果右子树非空则输出一个逗号加上右子树; 最后再输出一个右括号。当然, 如果左、右子树都是空的, 就没必要输出括号和括号内的。

由上面的分析可以看出, 以广义表形式输出的过程是前序遍历过程经过一点点修改后得到的:

```

procedure print (Node);
begin
  if Node < > nil then
    write (Node^.data);
    if (Node^.left < > nil) or (Node^.right < > nil) then
      write ( "(" );
      print (Node^.left);
      if Node^.right < > nil then
        write ( "," );
        print (Node^.right);
      write ( ")" );
end;

```



### 三、求二叉树的深度

若一棵二叉树为空，则它的深度为0，否则它的深度定义为：它左子树和右子树深度的最大值+1。显然，二叉树的深度是递归定义的，因此，在求二叉树的深度时也可以递归地求：

```
function depth (Node): integer;
begin
    if Note = nil then return 0;
    depth1 ← depth (Note^.left);
    depth2 ← depth (Note^.right);
    if depth1 > depth2 then
        return depth1 + 1
    else
        return depth2 + 1;
end;
```

## 6.4 二叉搜索树

### 一、二叉搜索树的定义

二叉搜索树 (Binary search tree) 又称二叉排序树或二叉查找树，它或者是一棵空树，或者是一棵具有如下特性的非空二叉树：

- (1) 若它的左子树非空，则左子树中所有的结点的值都不大于根结点的值；
- (2) 若它的右子树非空，则右子树中所有的结点的值都不小于根结点的值；
- (3) 它的左、右子树分别是一棵二叉搜索树。

由于二叉搜索树的值是按左子树、根、右子树有序的，所以对一棵二叉搜索树进行中序遍历访问所有结点所得的序列是有序的。

### 二、二叉搜索树的查找

由二叉搜索树的定义，在二叉搜索树中查找一个结点值为K的结点的过程为：若二叉树为空，则查找失败，返回空指针；若当前根结点的值等于K，则查找成功，返回当前的根结点；若K小于当前结点的值，则说明要找的结点只可能在当前结点的左子树中，进入左子树查找；若K大于当前结点的值，则说明要找的结点只可能在当前结点的右子树中，进入右子树查找。显然，这个过程和前面的过程一样，也是递归调用的。其实现过程如下：

```
function searchKey (Node, K): PNode;
begin
    if Node = nil then return nil;
    if K = Node^.data then return Node;
    if K < Node^.data then return searchKey (Node^.left, K);
    if K > Node^.data then return searchKey (Node^.right, K);
```



end;

从算法中可以看出,这个算法递归时一旦返回,就再也不会再出现递归的调用,这种递归叫做末尾递归。末尾递归可以写成非递归的形式,这样可以节省栈所用的空间和时间。如上面的算法可以写成:

```
function searchKey (Node, K): PNode;
begin
while (Node <> nil) do
    if K = Node^.data then return Node;
    if K > Node^.data then Node←Node^.left;
    if K < Node^.data then Node← Node^.right;
return nil;
end;
```

在对二叉搜索树进行查找的过程中,  $K$  与结点比较的次数最小为一次,即树的根结点就是要查找的结点,最多为树的深度次,所以平均的查找次数要小于等于树的深度。理论上已经证明,平均查找次数为  $1 + 4\log_2 n$  次。若二叉树是理想平衡二叉树或接近理想平衡树,则查找次数大概为  $\log_2 n$  次。因此,对二叉搜索树的查找的复杂度在最好情况和平均情况下都是  $O(\log_2 n)$  级别的。但是,要是二叉搜索树退化成一条链或近似链的情况,即单支结点非常多,则深度会非常大,查找时的复杂度为  $O(n)$ 。由上面的分析可知,利用二叉搜索树虽然最坏情况下和单链表的复杂度相同,但一般情况下比单链表好得多。至于空间复杂度,显然只有结点存在时要分配内存,因此空间复杂度为  $O(n)$  的。

### 三、二叉搜索树的插入和生成

二叉搜索树的插入是指在一棵二叉搜索树中插入一个关键字,使它仍满足二叉搜索树的要求。

二叉搜索树的生成通常是从一棵空树开始,将关键字一个一个地加入树中,从而产生一个具有很多结点的二叉搜索树。

在插入结点时,若当前的树为空,则建立一个结点,将这个结点作为根,并使它的值为插入的关键字;若关键字小于当前根结点的值,则应该当关键字插入左子树中,否则应插入右子树中。

上面的算法是递归的,可以写为:

```
procedure insert (var Node; key);
begin
    if Node = nil then
        new (Node);
        Node^.data←key;
        Node^.left←nil;
        Node^.right← nil;
        return;
    if key < Node^.data then insert (Node^.left, key)
    else insert (Node^.right, key);
end;
```

同查找一样,这里的递归也是末尾递归,可以转化为非递归来做。和递归不同的是,非递归

时必须记下父结点的指针是多少,而不能单纯地对其父结点的某一个子结点的指针操作。其实现过程为:

```

procedure insert (var Node; key);
begin
    new (TmpNode); TmpNode^.data←key; TmpNode^.left, TmpNode^.right←nil;
    if Node = nil then Node←TmpNode;
    p←Node; {当前处理的结点指针}
    q←nil; {父结点记录指针,树的根结点无父结点}
    while (p <> nil)
        q←p; {接下来要进入子结点,改变父结点指针}
        if key < p^.data then p←p^.left else p←p^.right; {改变当前结点指针}
    if key < q^.data then q^.left←TmpNode else q^.right←TmpNode;
end;
    
```

对二叉搜索树的一次插入的复杂度主要集中在查找上,所以最好情况和平均情况下都是  $O(\log_2 n)$  的,而最坏情况下是  $O(n)$  的。

二叉搜索树生成的算法可写为:

```

function Create (n): PNode; {n 为结点数}
begin
    Node←nil;
    for i in [1, n]
        read (key);
        insert (Node, key);
    return Node;
end;
    
```

设要建立二叉搜索树的序列为: (63, 71, 6, 52, 85, 53, 82, 38, 11), 则二叉搜索树的插入过程如图 6-14 所示:

#### 四、二叉搜索树的删除

二叉搜索树的删除比插入要复杂些。我们分几种情况讨论:

若结点为叶结点: 直接将结点删除。即, 若要删除的是根结点, 则将树置空; 若删除的是结点 X 的左子树, 则设 X 的左子树为空树; 若删除的是结点 X 的右子树, 则设 X 的右子树为空树。如图 6-15 (a) 是一棵二叉搜索树, (b) 是它删除关键字 11 后的二叉树。

若结点为单分支结点: 如果一个结点只有左子树或右子树, 然后, 直接把它删除, 用它的左子树或右子树代替它原来的位置即可。如图 6-15 (b) 是一棵二叉搜索树, (c) 是它删除关键字 6 后的二叉树。

若结点为双分支结点: 删除双分支结点的思想是, 将它的中序遍历前趋复制到它的位置覆盖它, 然后删除原来中序前趋结点。因为结点既有左孩子又有右孩子, 所以从它的左孩子开始, 每次都往右孩子走, 直到走不了了, 最后走到的结点就是它的中序前趋 (为什么? 请读者自己思考)。显然, 它的中序前趋的右孩子为空树, 因此删除时只要用上面的叶结点或单分支结点的删除方法就可以了。图 6-15 (c) 是一棵二叉搜索树, (d) 是它删除关键字 63 后的子树。

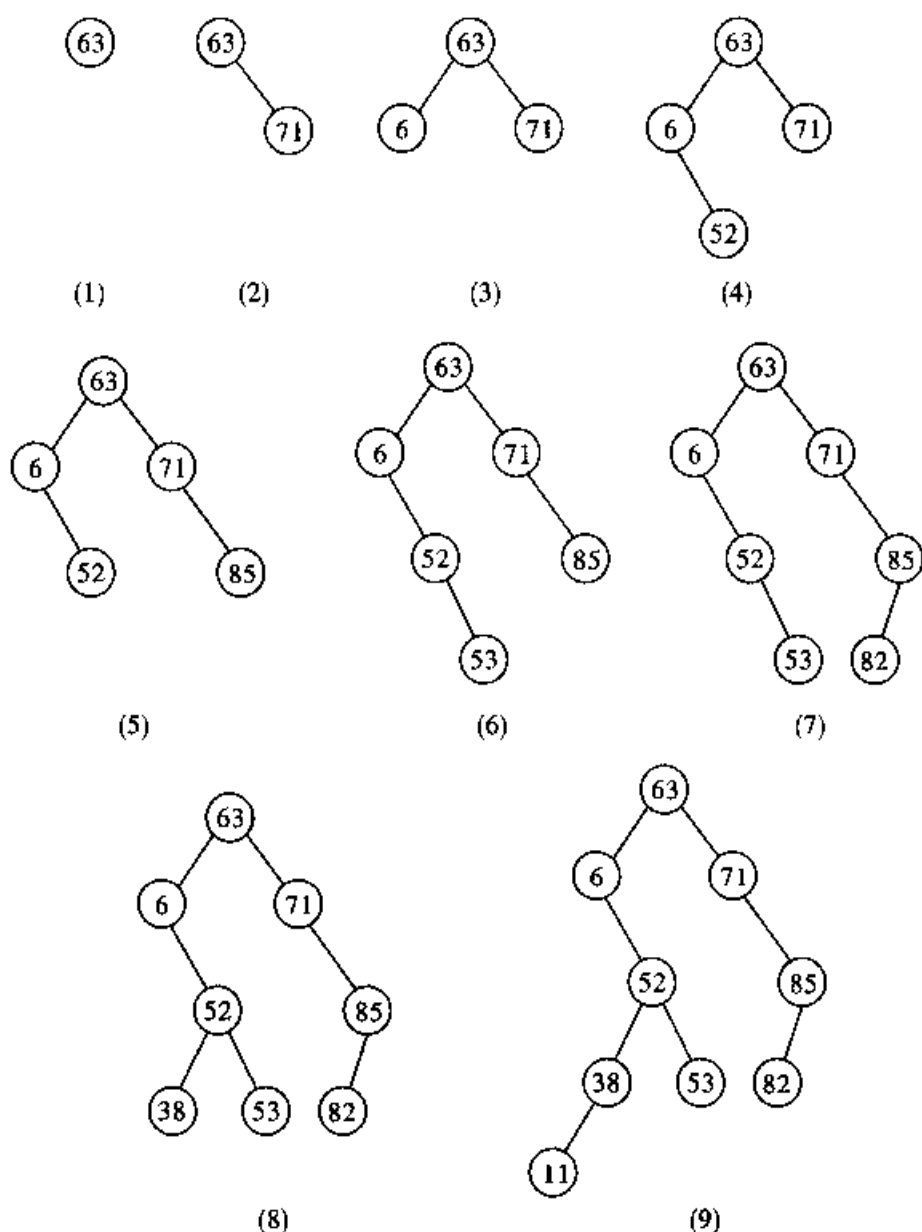


图 6-14 二叉搜索树的建立过程

综上所述，删除二叉搜索树中的一个关键字，可以如下实现：

```
function delete (var Node; key): boolean; {若成功，返回 true，否则返回 false}
begin
```

在 Node 中查找 key，使找到的结点为 p，其父结点为 q（此步骤可由查找的过程稍微变动一下得到，这里不再写出）。

```
if p = nil then return false;
```

```
if (p^.left = nil) and (p^.right = nil) then {实际上，这个过程可以放在左、右子树中有一个非空的里面一同处理}
```

```
if p = Node 则 Node ← nil; return true;
```

```
if p = q^.left 则 q^.left ← nil 否则 q^.right ← nil
```

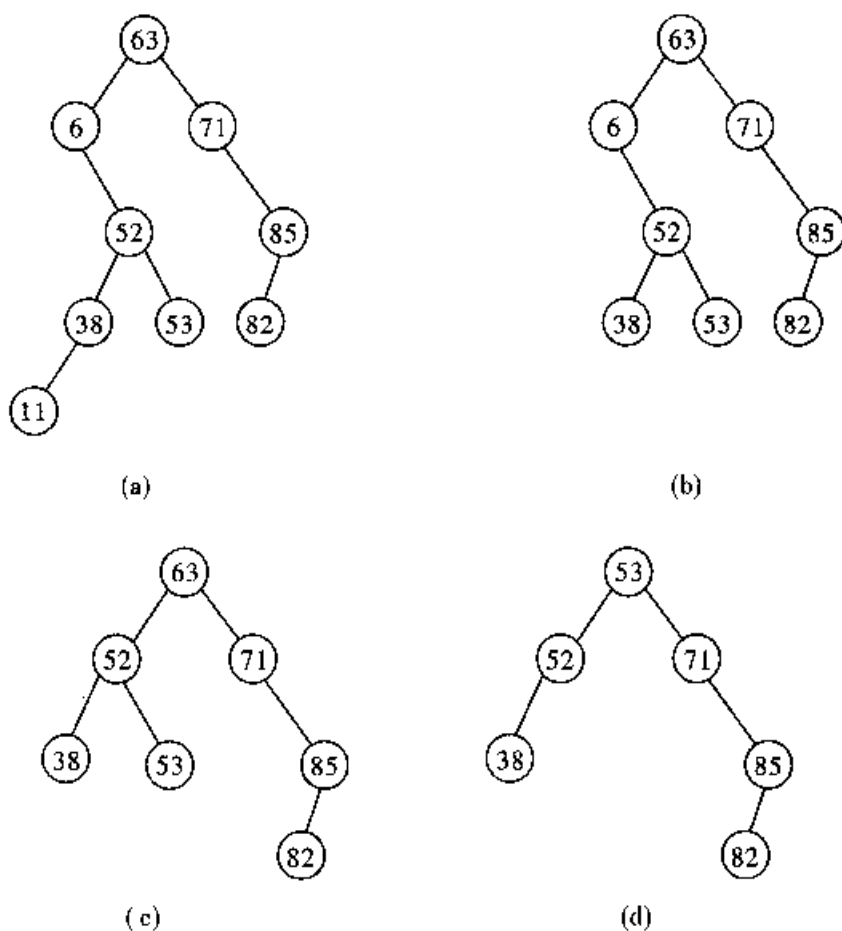


图 6-15 二叉树的删除

```

return true;
if (p^.left = nil) xor (p^.right = nil) then
    tmp ← p^.left (左子树非空) 或 p^.right (右子树非空)
    if p = Node then Node ← tmp
    if p = q^.left then q^.left ← tmp else q^.right ← tmp
    return true;
if (p^.left < > nil) and (p^.right < > nil)
    tmpq ← p; tmpp ← p^.left;
    while (tmpp^.right < > nil)
        tmpq ← tmpp; tmpp ← tmpp^.right;
    p^.data ← tmpp^.data;
    if tmpp = p^.left then p^.left ← tmpp^.left else tmpq^.right ← tmpp^.left;
end;

```

和对二叉搜索树的一次插入操作一样，二叉搜索树的一次删除操作的复杂度主要集中在查找上，所以最好情况和平均情况下都是  $O(\log_2 n)$  的，而最坏情况下是  $O(n)$  的。



## 6.5 哈夫曼树

### 一、基本术语

在一棵树中,若存在一个序列  $k_1, k_2, k_3, \dots, k_p$ , 使得  $k_i$  与  $k_{i+1}$  是有边相连的结点 (即父亲与孩子的关系), 则称序列  $k_1, k_2, k_3, \dots, k_p$  是一条从  $k_1$  到  $k_p$  的路径, 其中, 路径中所经过的边数  $p-1$  称为路径长度。在  $k_1$  到  $k_p$  的所有路径中, 长度最短的一条叫做最短路径, 最短路径的长度叫做最短路径长度。

在许多应用中, 常常将树中的结点赋上了一具有某种意义的数, 称为权。结点的带权路径长度指结点的权与该结点到根结点的最短路径长度的乘积。

树的带权路径长度, 是指所有叶结点的带权路径长度之和。通常记为:

$$WPL = \sum_{i=1}^n w_i l_i$$

其中  $n$  表示叶结点数,  $w_i$  和  $l_i$  分别表示叶结点  $k_i$  的权值和  $k_i$  到根的最短路径长度。

### 二、哈夫曼树

哈夫曼 (Huffman) 树又称最优二叉树, 它是  $n$  个带权叶结点构成的所有二叉树中, 带权路径长度最小的二叉树。因为构造这种树的算法最早是由哈夫曼于 1952 年提出的, 所以被称为哈夫曼树。

例如, 有四个权分别为 2, 5, 6, 8 的叶结点, 它们可以构成不同的二叉树, 图 6-16 中给出了几种, 他们的带权路径长度分别为:

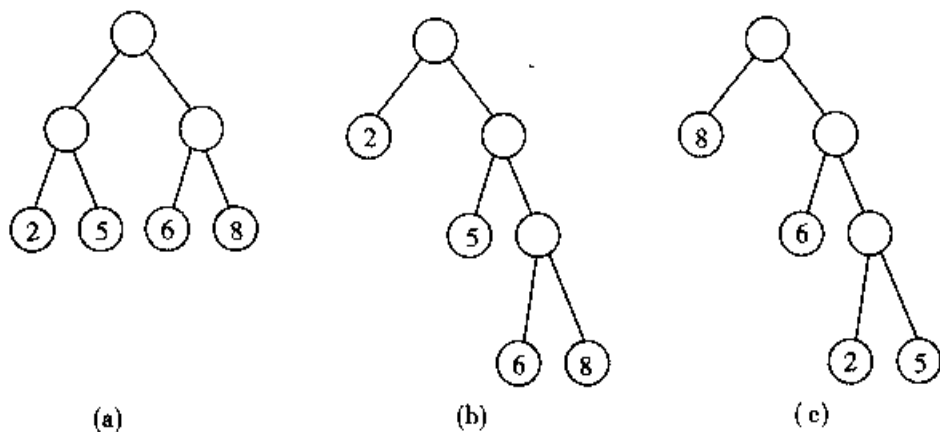


图 6-16 由四个叶结点构成的不同的二叉树

(a)  $WPL = 2 * 2 + 5 * 2 + 6 * 2 + 8 * 2 = 42$

(b)  $WPL = 2 * 1 + 5 * 2 + 6 * 3 + 8 * 3 = 54$

(c)  $WPL = 8 * 1 + 6 * 2 + 2 * 3 + 5 * 3 = 41$

可以看出, (c) 的 WPL 是最小的。稍后可知, (c) 就是哈夫曼树。

由上面的例子可以看出, 树的深度小的不一定带权路径长度小, 只有权大的结点放在尽量靠



根的层才能得到较优值。

### 三、构造哈夫曼树

构造哈夫曼树的算法是由哈夫曼提出的，所以称之为哈夫曼算法。具体过程如下：

(1) 根据  $n$  个权值  $\{w_1, w_2, w_3, \dots, w_n\}$  对应的  $n$  个结点构成  $n$  棵二叉树的森林  $F = \{T_1, T_2, T_3, \dots, T_n\}$ ，其中每棵二叉树  $T_i$  ( $1 \leq i \leq n$ ) 都有且仅有一个权值为  $w_i$  的根结点，其左、右子树为空；

(2) 在森林  $F$  中选出两棵根结点的权值最小的树作为一棵新树的左、右子树，且置新树的附加根结点的权值为其左、右子树上根结点的权值之和；

(3) 从  $F$  中删除这两棵树，同时把新树加入  $F$  中；

(4) 重复 (2) 和 (3)，直到  $F$  中只有一棵树为止，此树便是哈夫曼树。

图 6-17 中是用 (2, 5, 6, 8) 构造哈夫曼树的过程。

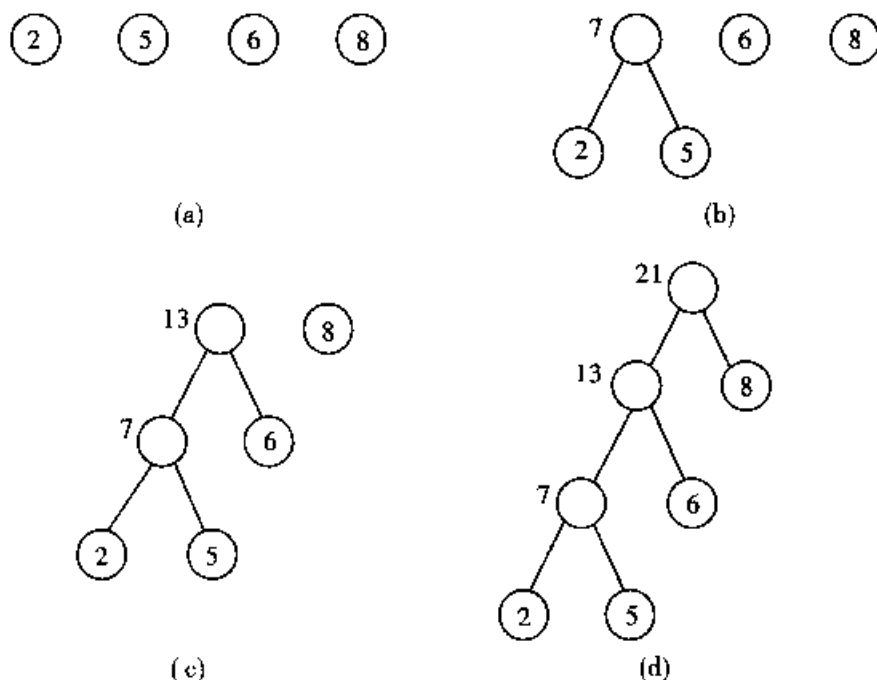


图 6-17 哈夫曼树的构造过程

在第二步中，若选权最小的有多于一种选法（如第二大的值和第三大的值相等），则选取不同的值就会对应不同的哈夫曼树；若左右子树摆放的顺序不同，也会产生不同的哈夫曼树。但是，不管选择或摆放的顺序是怎样，所构造出来的哈夫曼树的带权路径长度是不会变的。如图 6-16 (c) 和 6-17 (d) 都是由 {2, 5, 6, 8} 所产生的哈夫曼树，其带权路径长度都是  $2 \times 3 + 5 \times 3 + 6 \times 2 + 8 \times 1 = 41$ 。

### 四、带权路径长度与合并费用和

在哈夫曼树的构造中，如果合并两个权值最小的树的权值分别为  $a$  和  $b$ ，而合并这两棵树的费用记为  $a+b$ ，则构造哈夫曼树所用的费用与哈夫曼树的带权路径长度相等。

显然，对于一个权值为  $x$  的结点  $X$ ，在与它的兄弟合并时要累加权值，而它父亲与其兄弟合并时又计算了一遍，然后父亲的父亲又计算一遍……也就是说，当它是第  $L$  层时，它的权值就计算了  $L-1$  遍，刚好等于根结点到它的最短路径长度。所以带权路径长度与合并费用和是相等的。

## 五、哈夫曼码

哈夫曼树的应用较广，哈夫曼码是其中非常重要的一种应用例子。

在电报通讯中，电文是以二进制的 0、1 序列传送的。在发送端需要将电文中的字符序列转换成二进制序列（即编码），在接收端又需要把接收二进制序列转换成对应的字符序列（即译码）。

最简单的二进制编码方式就是等长编码。如，假定电文中只有 A、B、C、D、E 这五种编码，在进行等长编码时，它们至少需要三位二进制来表示，可以依次编码为：000、001、010、011、100。若用这六个字符作为六个叶子结点，生成一棵二叉树，使二叉树的每个结点的左、右分支分别用 0、1 编码，从根结点到叶结点所经过的分支的 0、1 编码序列等于叶结点的二进制编码，则对应的编码二叉树如图 6-18 所示。

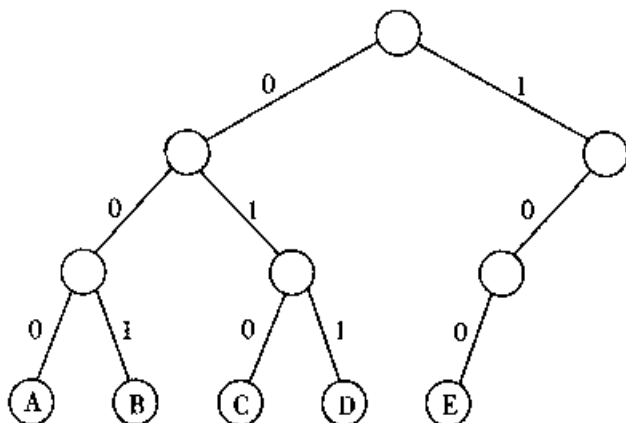


图 6-18 编码二叉树

显然，在电文中每个字母出现的频率（次数）一般是不同的。在一份电文中，设这五个字符出现的频率分别为  $c_i$ ，则这份电文的总长度为：

$$\text{Len} = \sum_{i=1}^n c_i l_i$$

其中  $n$  表示电文中的字符种数， $l_i$  表示第  $i$  种字符的编码长度。

若在上面的电文中，字符 A、B、C、D、E 分别为 2、5、6、8、12，则可算出此电文的总长度：

$$\text{Len} = \sum_{i=1}^n (c_i \times 3) = 3 \times (2 + 5 + 6 + 8 + 12) = 99$$

所以，当采用等长编码时，编码的总长度为 99。

怎样缩短传送电文的总长度，从而节省时间呢？可以想到，若采用不等长编码，让出现频率高的字符具有短的编码，出现频率低的字符具有长的编码，这样有可能缩短传送文件的长度。采用不等长编码必须避免译码的二义性或多义性。假设用 0 表示字符 A 的编码，01 表示 B 的编码，当接收到编码串 01……时，是用 0 译出 A 还是把 0 留着和 1 组成 01 译出 B 就是一个很大的问题，这样就产生了二义性。因此，若对一个字符集采用不等长编码，则要求字符集中任何字符的编码都不能是另一个字符编码的前缀。符合这个要求的编码叫做前缀编码。显然，等长编码是前缀编码。

为了使不等长编码为前缀编码，可以用该字符集中的字符作为叶结点构造一棵编码二叉树。将每个字符出现的频率作为对应结点的权值，这样，树的带权路径长度就是电文编码后的长度。

为了使电文的长度尽可能短,可以使用哈夫曼树。在上面的例子中,由 {2、5、6、8、12} 所生成的哈夫曼树如图 6-19 所示,由此,可以给字符以相应的编码: A: 000, B: 001, C: 01, D: 10, E: 11, 电文的最短传送长度为:

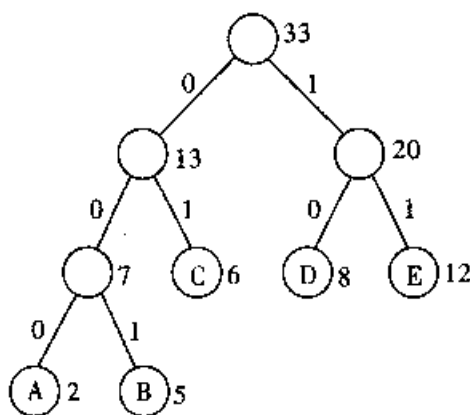


图 6-19 编码哈夫曼树

$$\text{len} = \text{WPL} = 2 * 3 + 5 * 3 + 6 * 2 + 8 * 2 + 12 * 2 = 73$$

显然, 这比原来的等长传送要短得多。

## 6.6 树的存储结构和运算

这里所指的树是指度不小于 2 的树, 在有的书上称为多叉树或多元树。下面我们均以三叉树为例对其进行讨论。

### 一、树的存储结构

对树的存储通常采用如下三种表示:

#### 1. 标准形式

在这种表示中, 树中的每个结点除了包含有存储数据元素的值域外, 还包含有三个指针域, 用来分别指向三个孩子结点, 或者说, 用来分别链接三棵子树。若结点采用动态产生, 则结点类型和指向结点的指针类型可定义为:

```

type
  PNode = ^TNode;
  TNode = record
    data : elementType;
    children : array [1..3] of PNode;
  end;
  
```

其中 children [1]、children [2]、children [3] 分别存储三个孩子结点的指针域。

#### 2. 广义标准形式

广义标准形式是在标准形式的每个结点中增加一个指向其双亲结点的指针域。

#### 3. 二叉树形式

这种表示指首先将树转换为对应的二叉树形式，然后再采用二叉链表存储这棵二叉树。其转换方法将在下一节中讨论。

## 二、树的运算

树的运算包括建立树的存储印象、遍历等。

### 1. 建立树的存储印象

假定给出树的广义表要求建立树的存储结构采用标准形式。

如图 6-20，其广义表表示为：

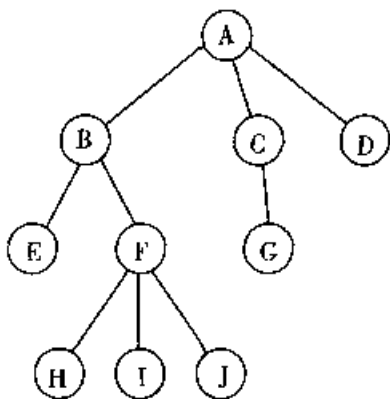


图 6-20 三叉树

A (B (E, F (H, I, J), C (G), D)

在树的生成算法中，需要设置两个栈：一个用来存储指向根结点的指针，以便将孩子结点链向双亲结点；另一个用来存储待链接的孩子结点的序号，以便能正确地链接到双亲结点的指针域。假定这两个栈分别用 S 和 P 表示，则树的生成算法可以写成如下的形式：

```
procedure buildtree;
```

```
begin
```

```
  read (ch);
```

```
  while (ch < > '@') do
```

```
    begin
```

```
      case ch of
```

```
        'A' .. 'Z': 新建结点，值域的值 of ch，左右子树指针为空；
```

如果这是第一个结点，那么它是根结点，否则将当前结点作为 S 栈顶结点的子树，其序号由 P 栈顶元素决定。

'('：将当前的结点插入 S 栈顶；将 1 插入 P 栈顶，表示接下来要链到其父结点的第一个指针域。

```
        ')': S、P 栈顶元素出栈。
```

```
        ',': 将 P 栈顶元素加 1，表示下一次处理的是其父结点的下一个指针域。
```

```
      end;
```

```
    read (ch);
```

```
  end;
```

end;

稍加分析即可发现,其实这里的操作和二叉树的非常类似,只是这里的结点数多一点,不再局限于左子树和右子树。

## 2. 遍历树

树的遍历分为深度优先遍历(有的书上也叫先根遍历)和广度优先遍历(也叫按层遍历),还有后根遍历,但第三种应用得不多。

深度优先遍历是指首先访问其根结点,然后从左到右递归地访问每一棵子树。如对图 6-20 中的树进行深度优先遍历的结果为:

A, B, E, F, H, I, J, C, G, D

仿照二叉树的先序遍历,可以给出三叉树的深度优先遍历的算法框架:

```
procedure preorder (Node);
begin
    if Node <> nil then
        write (Node^.data);
        for i := 1 to 3 do
            preorder (Node^.children [i]);
end;
```

广度优先遍历是指首先访问第一层的结点(即根结点),然后从左到右访问第二层的结点,然后第三层、第四层,直到所有的结点都访问完毕。如对图 6-20 中的树进行广度优先遍历的结果为:

A, B, C, D, E, F, G, H, I, J

在对树进行广度优先遍历时,需要设置一个队列,开始时队列为空。如果树为空树,则算法结束,否则将根结点插入队列中。以后的每一步,都取出队首元素,访问它,并从左到右将其子树插入队列中。这样所遍历的结果就是按广度优先遍历访问的结果。

广度优先遍历的算法框架可写为:

```
procedure layerorder (Node);
begin
    若 Node = nil 则 return;
    SteNull (Q);
    insert (Node, Q);
    repeat
        P ← Q.Pop;
        write (P^.data);
        for i := 1 to 3 do
            if P^.children [i] <> nil then
                insert (P^.children [i], Q);
        until Null (Q) = true;
    end;
```

## 6.7 树、森林与二叉树的转换

### 一、树转换成二叉树

树转换成二叉树的方法是：将树中每个结点的第一个孩子结点转换为二叉树中对应结点的左孩子，将第二个孩子结点转换为第一个孩子结点的右孩子，将第三个孩子结点转换成第二个孩子结点的右孩子，依此类推。这样，二叉树中的左孩子，实际就是树中的第一个孩子，而二叉树中的右孩子，则对应树中的右兄弟；显然这个转换是可逆的。例如，图 6-21 (a) 是一棵二叉树，转换成二叉树后如图 6-21 (b) 所示。

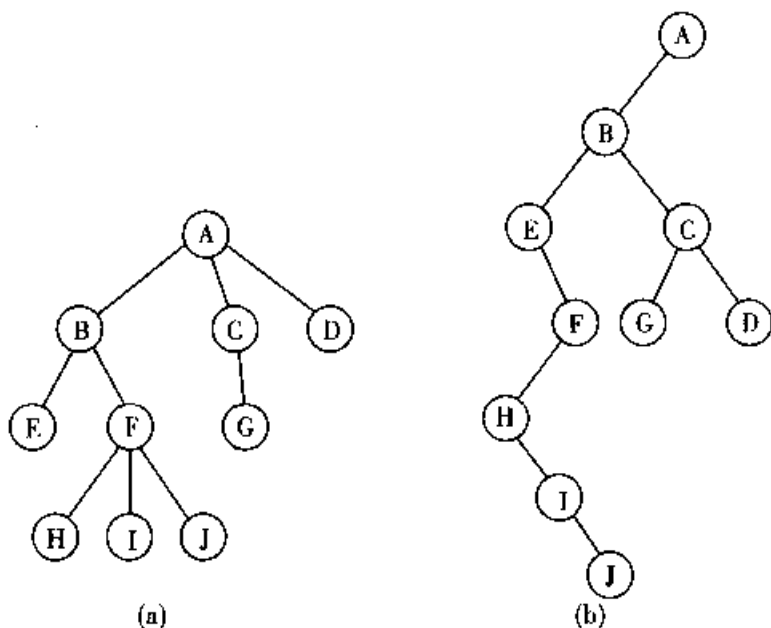


图 6-21 多叉树和其对应的二叉树

### 二、森林转换成二叉树

森林转换成二叉树，首先要将森林转换成树。

森林转换成树的方法是：增加一个虚拟结点  $V$ ，将森林中所有的树的根链接到虚拟结点上作为它的孩子结点。如图 6-22 (a) 中的森林转换成树如图 6-22 (b) 所示。

将森林转换成树后，再将树转换成对应的二叉树即可。

## 6.8 最近公共祖先

最近公共祖先 (Least Common Ancestors, LCA) 是树中一个基本问题，问题的表述如下：

LCA 问题：给出一棵有根树  $T$ ，对于任意两个结点  $u$  和  $v$ ，求出  $LCA(T, u, v)$ ，即离根最近的结点（或者说离  $u$  和  $v$  最近的结点） $r$ ，使得  $r$  同时是  $u$  和  $v$  的祖先。

最近公共祖先可以看作一种询问回答式的问题。解决这种题目有两种方式：一种是先进行充

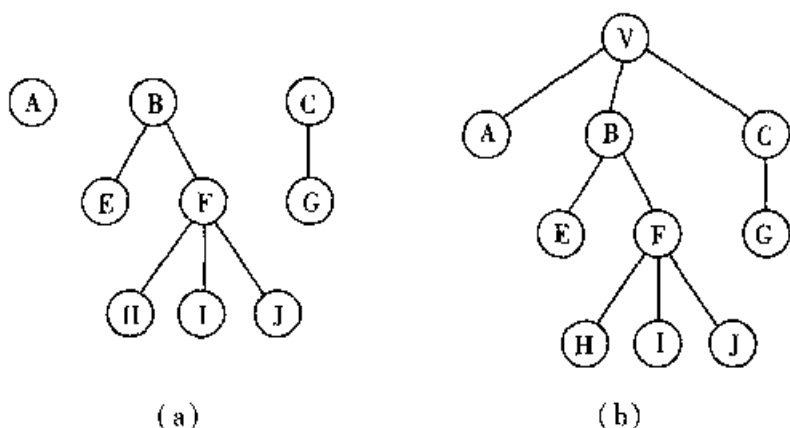


图 6-22 森林和对应的树

分的预处理, 然后每次回答询问的问题时只需要少量的时间, 这种算法叫做在线算法 (online - algorithm); 另一种是先将所有的询问收集 (读入), 然后将所有的回答一起完成回答。

最朴素的在线算法是: 将  $u$  到根结点路径上的结点进行标记, 这些结点都是  $u$  的祖先结点。然后从  $v$  开始往上枚举, 那么第一个发现被标记的结点  $x$ , 那么就是  $u$  和  $v$  的最近公共祖先。

显然这个算法每次询问的时间复杂度可能达到  $O(n)$ 。而至多有  $O(n^2)$  种不同的询问, 因此最坏情况可能达到  $O(n^3)$  的时间复杂度。

令  $L(u)$  为  $u$  的层数 (到根结点的距离),  $Fa(u, d)$  表示到  $u$  的距离为  $d$  的祖先结点。考虑一种特殊情况即  $L(u) = L(v)$  时, 设  $Fa(u, 1) = Fa(v, 1) = x$  为  $u$  和  $v$  的最近公共祖先。那么有这样一个性质: 对于所有  $k < 1$ , 有  $Fa(u, k) \neq Fa(v, k)$ ; 对于所有  $k \geq 1$ , 有  $Fa(u, k) = Fa(v, k)$ 。由于函数的连续性, 所以可以利用二分寻找第一个  $Fa(u, k) = Fa(v, k)$ 。因此, 现在需要能在较低的时间复杂度中计算  $Fa(u, d)$ 。

由于  $Fa(u, d)$  的个数有可能达到  $O(n^2)$  级别, 所以不可能在预处理中将所有的  $Fa(u, d)$  求出来。可以只保存其中的部分  $Fa(u, d)$ ; 令  $f(u, k) = Fa(u, 2^k)$ 。利用  $f(u, k)$  能够在  $O(\log_2 n)$  的时间复杂度下查找满足  $L(u) = L(v)$  的  $u, v$  两点的最近公共祖先  $x$ :

```

procedure getLCA (u, v);
begin
  if u = v then begin
    x ← u; exit;
  end;
  k ← 0;
  while k ≥ 0 do begin
    if f[u, k] ≠ f[v, k] then begin
      i ← f[u, k];
      j ← f[v, k];
    end;
    dec (k);
  end;
  x ← f[u, 0];

```



end;

上面的代码事实上是利用  $f[u, d]$  函数, 让  $u$  和  $v$  两个结点尽量往上爬 (寻找父结点), 最后找到满足  $Fa[u, l] \neq Fa[v, l]$  的最大值  $l$ , 那么  $Fa[u, l+1]$  便是  $u$  和  $v$  的最近公共祖先。

对于函数  $f[u, d]$  可以利用递推的方法求得:  $f[u, d] = f[f[u, d-1], d-1]$ 。递推一次的时间复杂度为  $O(1)$ , 而  $f[u, d]$  最多有  $O(n \log_2 n)$  种状态, 因此计算所有  $f[u, d]$  的时间复杂度为  $O(n \log_2 n)$ 。

下面考虑  $L(u) \neq L(v)$  的情况: 不妨设  $L(u) < L(v)$ , 那么有  $LCA(u, v) = LCA(Fa(u, L(u) - L(v)), v)$ 。因此只需要求出  $Fa(u, L(u) - L(v))$  即可:

```

procedure getLCA2 (u, v);
begin
  k ← ⌈log2 L(u)⌉
  while L(u) ≠ L(v) do begin
    if L(u) + 2k ≤ L(v) then
      u ← f(u, k)
    dec(k);
  end;
  x ← f[u, 0];
end;
```

同样, 这一步骤也可以在  $O(\log_2 n)$  的时间复杂度下完成。因此这个方法可以在  $O(n \log_2 n) - O(\log_2 n)$  的时间复杂度下完成在线 LCA 问题。利用这种算法也可以执行一些特殊的操作, 比如将两棵树的根进行合并。合并操作同样能够在  $O(\log_2 n)$  的时间复杂度下完成。具体的实现方法请读者自行思考。

下面介绍另一种离线的 LCA 算法, 它能够在  $O(n\alpha(n) + Q)$  的时间复杂度下回答所有 LCA 询问。其中  $\alpha(n)$  是 Ackermann 函数的反函数, 可以近似地看成小于 5。而  $Q$  是询问的次数。

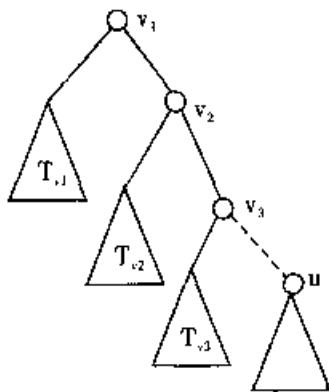


图 6-23 遍历结点

考虑深度优先遍历整棵树, 遍历到某个结点  $u$ , 如图 6-23 所示:

其中  $v_1, v_2, \dots, v_m$  表示根结点到  $u$  的路径上的结点, 显然它们是  $u$  所有的祖先结点。图中的  $T_i$  是一个已经遍历完毕的结点的集合, 集合中的每个结点  $w$ , 满足在  $w$  所有未出栈的祖先结点 (未出栈的结点中即其子树中仍有未遍历的结点), 离其最近的结点为  $x$ 。那么显然有对于  $T_i$  中的



所有点  $w$ , 都有  $LCA(u, w) = v_i$ 。因此对于有关于  $u$  的所有询问  $LCA(u, w)$ , 只要  $w$  已经被遍历过, 且知道  $w$  属于  $T_x$  集合, 就能够得知  $LCA(u, w) = x$ 。可以通过如下的代码回答所有的询问:

```

procedure Dfs (u);
begin
  T[u] ← {u}; // 将  $T_u$  集合初始化
  for u 的每个孩子 v do begin
    Dfs (v);
    T[u] ← T[u] ∪ T[v];
    T[v] ← ?; // v 和其子树全部被遍历, 因此将  $T_v$  中的结点加入到  $T_u$  中,  $T_v$  清空。
  end;
  mark[u] ← true; // 表示结点 u 已经被遍历完毕。
  for 所有关于 u 的询问 LCA(u, v) do
    if mark[v] then // 如果 v 已经遍历, 那么  $LCA(u, v) = x$ , 其中 v 属于  $T_x$ 
      LCA(u, v) = Find(v); // Find(v) 是用来查找 x 的。
  end;

```

在上述代码中, 要对  $T$  集合进行合并和元素的查询工作, 这两种功能能够通过并查集这种数据结构来实现。并查集将在后面的章节进行介绍, 用它实现合并的均摊时间复杂度为  $O(\alpha(n))$ , 而查找为  $O(1)$ , 所以这个算法能在  $O(n\alpha(n) + Q)$  的时间复杂度下漂亮地解决本问题。

除此之外, LCA 问题还能够转化为 RMQ (Range Minimal Query) 解决, 在线算法的时间复杂度能够降到  $O(n) - O(1)$ 。具体的实现方法这里不再论述, 有兴趣的读者可以参照其他的文献。

## 6.9 树状数组

树状数组是一种高效的动态数据结构。从名称上看, 树状数组首先是数组, 但同时包含了树的思想。这就导致树状数组在操作上与普通数组有很多不同点, 当然还有随之而来的在效率上的飞跃。

先考虑一个简单的统计问题: 一维子序列动态求和。设该序列为  $a[1], a[2], \dots, a[n]$ , 要求提供两种操作:

1. 更新元素值: 将  $a[x]$  的某一位加上  $y$ ;
2. 子序列求和: 统计  $a[1] + a[2] + \dots + a[x]$  的和。

算法 1: 更新元素值时直接在原序列中作加法, 显然其时间复杂度为  $O(1)$ ; 子序列求和时直接将  $a[1] + a[2] + \dots + a[x]$  的和累计, 在最坏情况下, 子序列求和的时间复杂度为  $O(n)$ 。

算法 2: 增加辅助序列  $b$ , 其中  $b[i] = a[1] + a[2] + \dots + a[i]$ 。由于  $a[i]$  的更改影响  $b[i] \dots b[n]$ , 因此在最坏情况下更新元素值的时间复杂度为  $O(n)$ ; 而子序列求和的时间



复杂度仅为  $O(1)$ 。

以上两种算法,要么在更新元素值上耗费时间过长(算法1),要么在子序列求和上无法避免大量运算(算法2)。根据短板原理,木桶装水量取决于最短的一块木板的长度,因此算法1和算法2能装的“水”都太少了。

虽然无法推翻短板原理,但是可以通过巧妙的方式避开它:把木桶平放在地上,装一些水,使液面刚好达到最短的木板。这时,尝试把木桶向较长木板的一侧倾斜,你会发现,木桶能装更多的水了!

回到我们原来的问题,如何将“木桶”倾斜呢?这实际上就是将更新元素值和子序列求和的复杂度调和。让我们看看算法3:

算法3:增加序列  $C$ , 其中  $C[i] = a[i-2k+1] + \dots + a[i]$  ( $k$  为  $i$  在二进制形式下末尾0的个数)。由  $C$  数组的定义可以得出:

$$C[1] = a[1]$$

$$C[2] = a[1] + a[2] = c[1] + a[2]$$

$$C[3] = a[3]$$

$$C[4] = a[1] + a[2] + a[3] + a[4] = c[2] + c[3] + a[4]$$

$$C[5] = a[5]$$

$$C[6] = a[5] + a[6] = c[5] + a[6]$$

.....

更新元素值:

引理1:若  $a[k]$  所牵动的序列为  $C[p_1], C[p_2], \dots, C[p_m]$ , 且  $p_1 < p_2 < \dots < p_m$ , 则有  $l_1 < l_2 < \dots < l_m$  ( $l_i$  为  $p_i$  在二进制中末尾0的个数)。

证明:若存在某个  $i$  有  $l_i \geq l_{i+1}$ , 则由  $p_i - 2^{l_i+1} \leq k \leq p_i$ ,  $p_{i+1} - 2^{l_i+1} + 1 \leq k \leq p_{i+1}$  得

$$p_{i+1} - 2^{l_i+1} + 1 \leq k \leq p_i,$$

即

$$p_{i+1} \leq p_i + 2^{l_i+1} - 1 \quad (1)$$

而由  $l_i \geq l_{i+1}$ ,  $p_i < p_{i+1}$  可得

$$p_{i+1} \geq p_i + 2^{l_i} \quad (2)$$

(1)(2) 矛盾, 可知  $l_1 < l_2 < \dots < l_m$ , 即引理1成立。

定理1:若  $a[k]$  所牵动的序列为  $C[p_1], C[p_2], \dots, C[p_m]$ 。则  $p_1 = k$ , 而  $p_{i+1} = p_i + 2^{l_i}$  ( $l_i$  为  $p_i$  在二进制中末尾0的个数)。

证明:

因为  $p_1 < p_2 < \dots < p_m$  且  $C[p_1], C[p_2], \dots, C[p_m]$  中包含  $a[k]$ , 因此  $p_1 = k$ 。在  $p$  序列中,  $p_{i+1} = p_i + 2^{l_i}$  是  $p_i$  后最小的一个满足  $l_{i+1} > l_i$  的数(若出现  $p_i + x$  比  $p_{i+1}$  更小, 则  $x < 2^{l_i}$ , 与  $x$  在二进制中的位数小于  $l_i$  相矛盾)。  $p_{i+1} = p_i + 2^{l_i}$ ,  $l_{i+1} \geq l_i + 1$ 。由  $p_i - 2^{l_i+1} \leq k \leq p_i$  可知,  $p_{i+1} - 2^{l_i+1} + 1 \leq p_i + 2^{l_i} - 2 \times 2^{l_i} + 1 = p_i - 2^{l_i} + 1 \leq k \leq p_i \leq p_{i+1}$ , 故  $p_i$  与  $p_{i+1}$  之间的递推关系式为:

$$p_{i+1} = p_i + 2^{l_i}$$

因此定理1成立。

由此得出更改元素值的方法:若将  $\text{delta}$  添加到  $a[k]$ , 则  $C$  序列  $C[p_1], C[p_2], \dots, C$

$[p_m]$  ( $p_m \leq n < p_{m+1}$ ) 受其影响, 亦应该添加  $\delta$ 。例如在  $a[1] \dots a[9]$  中, 将  $a[3]$  添加  $\delta$ :

$$p_1 = k = 3$$

$$p_2 = 3 + 2^0 = 4$$

$$p_3 = 4 + 2^2 = 8$$

$$p_4 = 8 + 2^3 = 16 > 9$$

由此得出,  $C[3]$ ,  $C[4]$ ,  $C[8]$  亦应该添加  $x$ 。下面给出更新元素值的伪代码:

```
procedure update (k, delta);
begin
    p ← k;
    while (p ≤ limit) do
        C[p] ← C[p] + delta;
        p := p + lowbit (p);
    end;
```

其中  $\text{lowbit}(p)$  表示求 2 的 1 次方, 1 为  $p$  的二进制末尾 0 的个数。比较简单的求法是:

```
function lowbit (x): integer;
begin
    return x and (x xor (x - 1));
end;
```

讨论子序列求和问题。

类似于更新元素值, 子序列求和可以转化为求由  $a[1]$  开始的序列  $a[1] \dots a[k]$  的和  $S$ 。而在树状数组中求  $S$  十分简单: 根据  $C[k] = a[k - 2l + 1] + \dots + a[k]$  ( $l$  为  $k$  在二进制数中末尾 0 的个数)。我们从  $k_1 = k$  出发, 按照  $k_{i+1} = k_i - 2l_k$  ( $l_k$  为  $k_i$  在二进制数中末尾 0 的个数) 递推  $k_2, k_3, \dots, k_m$  ( $k_m + 1 = 0$ )。由此得出  $S = C[k_1] + C[k_2] + C[k_3] + \dots + C[k_m]$ 。例如, 计算  $a[1] + a[2] + a[3] + a[4] + a[5] + a[6] + a[7]$ :

$$k_1 = 7$$

$$k_2 = k_1 - 2^{l_1} = 7 - 2^0 = 6$$

$$k_3 = k_2 - 2^{l_2} = 6 - 2^1 = 4$$

$$k_4 = k_3 - 2^{l_3} = 4 - 2^2 = 0$$

即  $a[1] + a[2] + a[3] + a[4] + a[5] + a[6] + a[7] = c[7] + c[6] + c[4]$ 。由此我们可以写出子序列求和的伪代码:

```
function getsum (k): integer;
begin
    p ← k;
    sum ← 0;
    while (p > 0) do
        sum ← sum + C[p];
```



```

    p ← p - lowbit (p);
  return sum;
end;
```

以上就是树状数组的基本介绍。

树状数组的更新元素值操作是针对点进行的,也就是说不能一次更新一段元素值,在这一点上比线段树弱。但是树状数组时空效率较高,并且较容易编写,因此在面临树状数组和线段树都可解决的问题时当然要优先选择树状数组。

## 6.10 并查集

集合是一个数学概念,一般地,某些指定的对象集在一起就成为一个集合。集合中的元素满足:

1. 确定性:按照明确的判断标准给定一个元素或者在这个集合里,或者不在,不能模棱两可;
2. 互异性:集合中的元素没有重复;
3. 无序性:集合中的元素没有一定的顺序(通常用正常的顺序写出)。

如果不考虑罗素悖论的话,以上对集合的定义是很精确的。但对于计算机科学来讲,要的不仅仅是精确,而是连续+离散=具体(continuous + discrete = concrete)。对于一般的应用来说,集合至少需要提供两个操作:

1. 建立集合;
2. 查找某个元素是否在一给定集合内;
3. 合并两个集合。

我们首先考虑集合的表示方法,然后在此基础上讨论其操作。假设集合中有  $n$  个元素,分别用 1 到  $n$  表示。

表示方法 1: 建立标记数组  $A$ ,  $A[i]$  表示元素  $i$  属于的集合的标记。建立集合使用以下操作:

```

procedure makeset (x);
begin
```

```
    A [x] ← x;
```

```
end;
```

每次查找时判断  $A[i]$  是否是给定集合的标记:

```
function find (x): integer;
```

```
begin
```

```
    return A [x];
```

```
end;
```

每次合并时进行下列操作:

```
procedure union (x, y);
```

```
begin
```

```
    p ← A [x];
```



```
for i: = 1 to n do
```

```
  if A [ i ] = p then A [ i ] ← A [ y ];
```

```
end;
```

也就是将  $x$  所在集合的所有元素的集合标记都改为  $y$  所在的集合，复杂度是  $O(n)$  的。虽然查找效率特别高，但同时合并的效率又太低了。

表示方法 2：用一个结点对应一个元素，在同一个集合中的结点串成一条链表就得到了单链表的表示方法。在集合中我们以单链表的第一个结点作为集合的代表元。于是每个结点  $x$  ( $x$  也是元素的编号) 应包含这些信息：指向代表元即表首的指针  $head[x]$ ，指向表尾的指针  $tail[x]$ ，下一个结点的指针  $next[x]$ 。集合建立过程设计如下：

```
procedure makeset (x)
```

```
begin
```

```
  head [ x ] ← x;
```

```
  tail [ x ] ← x;
```

```
  next [ x ] ← nil;
```

```
end;
```

求代表元的算法设计如下：

```
function find (x): integer;
```

```
begin
```

```
  return head [ x ];
```

```
end;
```

前两个过程比较简单，合并两个集合的操作稍微复杂一点。我们要做的是将  $x$  所在链表加到  $y$  所在链表尾，然后  $y$  所在链表中的所有结点的代表元指针改指  $x$  所在链表的表首结点，如图 6-24 所示。

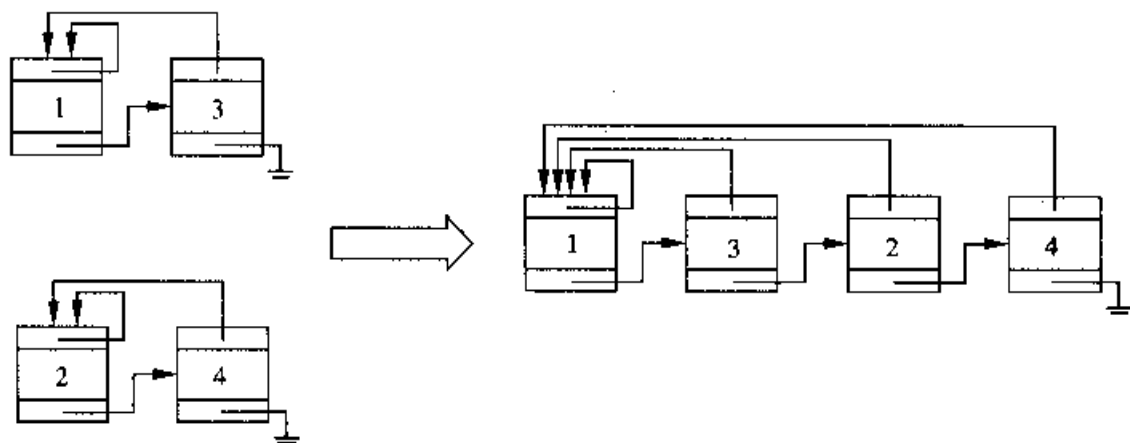


图 6-24 合并集合

合并操作的伪代码如下：

```
procedure union (x, y);
```

```
begin
```

```
  next [ tail [ head [ x ] ] ] ← head [ y ];
```

```

tail [ head [ x ] ] ← tail [ head [ y ] ];
p ← head [ y ];
while ( p < > nil ) do
    head [ p ] ← head [ x ]
    p ← next [ p ]
end;

```

现在我们来分析一下算法的时间效率。建立集合和查找元素都只需要  $O(1)$  的时间，而合并两个集合的时间效率与  $y$  所在链表的长度成线性关系。最坏情况下，即有操作序列  $\text{union}(n-1, n)$ ,  $\text{union}(n-2, n-1)$ ,  $\dots$ ,  $\text{union}(1, 2)$  时， $n-1$  次集合合并的时间复杂度为  $O(n^2)$ 。现在我们考虑如何减小合并时的复杂度。

我们想到合并链表时，我们可以用一种启发式的方法：将较短的链表合并到较长的链表上。为此每个代表节点中还需包含表的长度的信息，这可以在合并集合的时候进行更新。

我们来分析一下现在  $\text{union}(x, y)$  的时间复杂度。

首先我们给出一个固定对象  $x$  的代表元指针  $\text{head}[x]$  被更新次数的上界。由于每次  $x$  的代表元指针被更新时， $x$  必然在较小的集合中，因此  $x$  的代表元指针被更新一次后，集合至少含 2 个元素。类似地，下一次更新后，集合至少含 4 个元素，继续下去，当  $x$  的代表元指针被更新  $\log_2 k$  次后，集合至少含  $k$  个元素，而集合最多含  $n$  个元素，所以  $x$  的代表元指针至多被更新  $\log_2 n$  次。所以每次合并操作的时间复杂度至多为  $O(\log_2 n)$ 。

表示方法 3：集合的另一种更好的实现方法是用有根树来表示：每棵树表示一个集合，树中的结点对应一个元素。图 6-25 示出了一个分离集合的森林。

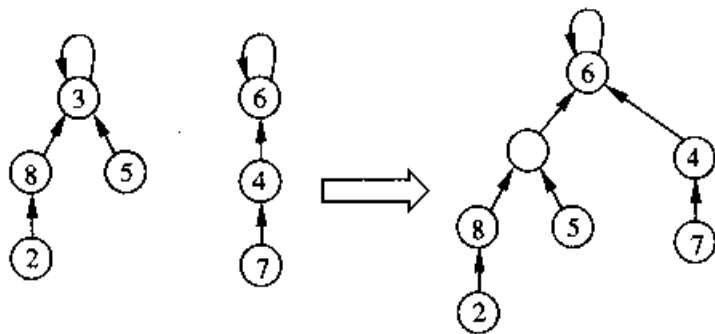


图 6-25 一个分离集合的森林

每个结点  $x$  包含这些信息：父结点指针  $p[x]$ ，树的深度  $\text{depth}[x]$ 。其中  $\text{depth}[x]$  将用于启发式合并过程。于是建立集合过程的时间复杂度依然为  $O(1)$ 。

```

procedure makeset (x);
begin
    p [ x ] ← x;
    depth [ x ] ← 0;
end;

```

用森林来表示集合的最大好处就是降低  $\text{union}(x, y)$  过程的时间复杂度。

```

procedure union (x, y);
begin

```

```
fx ← find (x);
```

```
fy ← find (y);
```

```
p [fx] ← fy;
```

```
end;
```

合并集合的工作只是将  $x$  所在树的根结点的父结点改为  $y$  所在树的根结点。在找到根结点之后, 这个操作只需  $O(1)$  的时间。而  $\text{union}(x, y)$  的时间效率决定于  $\text{find}(x)$  的快慢。

```
function find (x): integer;
```

```
begin
```

```
  if  $x \neq p[x]$  then return x
```

```
  else return find (p [x]);
```

```
end;
```

这个过程的时间与树的深度成线性关系, 因此其平均时间复杂度为  $O(\log_2 n)$ , 但在最坏情况下 (树退化成链表), 时间复杂度为  $O(n)$ , 因此有必要对算法进行优化。

第一个优化是启发式合并。在优化单链表时, 我们将较短的表链到较长的表尾, 在这里我们可以用同样的方法, 将深度较小的树指到深度较大的树的根上。这样可以防止树的退化, 最坏情况不会出现。于是  $\text{find}(x)$  的时间复杂度降为  $O(\log_2 n)$ , 其过程也要作相应改动。

```
procedure union (x, y);
```

```
begin
```

```
  fx ← find (x);
```

```
  fy ← find (y);
```

```
  if depth [fx] > depth [fy] then
```

```
    p [fy] ← fx
```

```
  else
```

```
    p [fx] ← fy;
```

```
    if depth [fx] = depth [fy] then depth [fy] ← depth [fy] + 1;
```

```
end;
```

然而算法的耗时主要还是花在  $\text{find}(x)$  上。

第二个优化是路径压缩。它非常简单而有效。如图 6-26 所示, 在  $\text{find}(1)$  时, 我们“顺便”将结点 1, 2, 3 的父结点均改为结点 4, 以后再调用  $\text{find}(1)$  时就只需  $O(1)$  的时间。

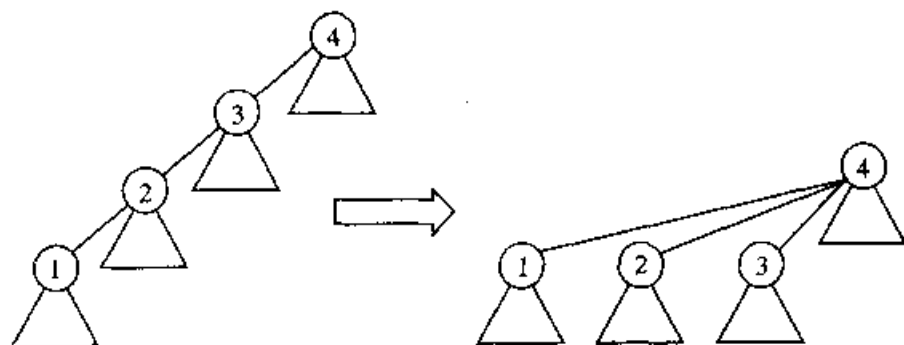


图 6-26 路径压缩



于是 find (x) 的代码改为:

```
function find (x): integer;
begin
  if x < > p [x] then p [x] ← find (p [x]);
  return p [x]
end;
```

该过程首先找到树的根, 然后将路径上的所有结点的父结点改为这个根。实现时, 递归的程序有许多栈的操作, 改成非递归会更快些。

```
function find (x): integer;
begin
  r ← x;
  while (r < > p [r]) do
    r ← p [r]
  while (x < > r) do
    q ← p [x];
    p [x] ← r;
    x ← q;
  return r;
end;
```

改进后的算法时间复杂度的分析十分复杂, 这里只给出结论: 改进后的查找元素和合并集合操作是  $O(\alpha(n))$  的, 其中  $\alpha(n)$  是阿克曼函数的反函数, 在可以想象的范围内都是小于等于 4 的, 因此可以认为是常数级别。

## 6.11 树的应用举例

**例题 6-5 树的重量。**

**问题描述:**

树可以用来表示物种之间的进化关系。一棵“进化树”是一个带边权的树, 其叶结点表示一个物种, 两个叶结点之间的距离表示两个物种的差异。现在, 一个重要的问题是, 根据物种之间的距离, 重构相应的“进化树”。

令  $N = \{1..n\}$ , 用一个  $N$  上的矩阵  $M$  来定义树  $T$ 。其中, 矩阵  $M$  满足: 对于任意的  $i, j, k$ , 有  $M[i, j] + M[j, k] \leq M[i, k]$ 。树  $T$  满足:

1. 叶节点属于集合  $N$ ;
2. 边权均为非负整数;
3.  $d_T(i, j) = M[i, j]$ , 其中  $d_T(i, j)$  表示树上  $i$  到  $j$  的最短路径长度。

如下矩阵  $M$  描述了一棵树。

$$M = \begin{bmatrix} 0 & 5 & 9 & 12 & 8 \\ 5 & 0 & 8 & 11 & 7 \\ 9 & 8 & 0 & 5 & 1 \\ 12 & 11 & 5 & 0 & 4 \\ 8 & 7 & 1 & 4 & 0 \end{bmatrix}$$

树的重量是指树上所有边权之和。对于任意给出的合法矩阵  $M$ ，它所能表示树的重量是唯一确定的，不可能找到两棵不同重量的树，它们都符合矩阵  $M$ 。你的任务就是，根据给出的矩阵  $M$ ，计算  $M$  所表示树的重量。图 6-27 是上面给出的矩阵  $M$  所能表示的一棵树，这棵树的总重量为 15。

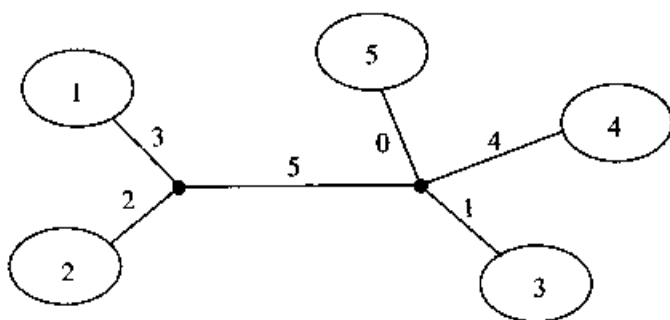


图 6-27 示例图

输入：

输入数据包含若干组数据。每组数据的第一行，是一个整数  $n$  ( $2 < n < 30$ )。其后  $n-1$  行，给出的是矩阵  $M$  的一个上三角（不包含对角线），矩阵中所有元素是不超过 100 的非负整数。输入数据保证合法。

输入数据以  $n=0$  结尾。

数据不超过 100 组。

输出：

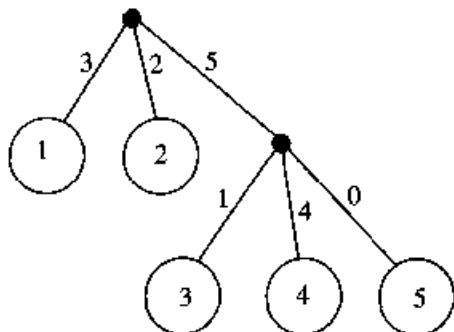
对于每组输入，输出一行，一个整数，表示树的重量。

样例

Weight. in	weight. out
5	15
5 9 12 8	71
8 11 7	
5 1	
4	
4	
15 36 60	
31 55	
36	
0	

分析：

本题中给出的实际是一棵无根树，对于这类题，我们一般将无根树转化为有根树，即任取一个结点作为根。当然，这题中，用一个非分支结点作为根不便于处理问题，因此，我们令一个分支结点为根。如图 6-27 中的根就可以表示为：



对于连到同一个结点  $P$  的两个叶结点  $A$  和  $B$ ，则  $M[A, B] = M[A, P] + M[P, B]$ ，如果  $C$  是另一个叶结点，则  $M[P, C] = M[A, C] - M[A, P] = M[B, C] - M[B, P]$ ，若对于所有的  $C$  都满足，则可以认为  $A, B$  是连到  $P$  上的。当确定  $A, B$  是同连到同一个结点上时，对于  $A, B$  分别考虑就没有必要了，以后可以考虑  $P$  而不考虑  $A, B$ 。这样，可以删除  $A$  和  $B$ ，同时认为增加了一个叶结点  $P$ 。

由于题中已说明了树是肯定存在的，所以这样做肯定可以做到最后只剩下一个结点。

源程序：

```
program weight;
```

```
const
```

```
  inf = 'weight.in';
```

```
  outf = 'weight.out';
```

```
  maxn = 30;
```

```
var
```

```
  n : integer;
```

```
  m : array [1..maxn, 1..maxn] of integer;
```

```
  used : array [1..maxn] of boolean;
```

```
  ans : integer;
```

```
function init : boolean;
```

```
var
```

```
  i, j : integer;
```

```
begin
```

```
  read (n);
```

```
  init := n > 0;
```

```
  for i := 1 to n - 1 do
```

```

    for j : = i + 1 to n do
    begin
        read (m [i, j]);
        m [j, i] : = m [i, j];
    end;
end;

procedure prepare;
begin
    fillchar (used, sizeof (used), 0);
    ans : = 0;
end;

function canJoin (a, b : integer; var delta : integer) : boolean;
var
    first : boolean;
    k : integer;
begin
    canJoin : = false;
    delta : = 0;
    first : = true;
    for k : = 1 to n do
        if not used [k] and (k < > a) and (k < > b) then
            if first then
                begin
                    delta : = m [a, k] - m [b, k];
                    first : = false;
                end
            else
                if m [a, k] - m [b, k] < > delta then
                    exit;
            canJoin : = true;
        end;
    end;

procedure joinone;
var
    i, j, k : integer;
    delta : integer;
begin

```



```

for i : = 1 to n - 1 do
  if not used [i] then
    for j : = i + 1 to n do
      if not used [j] then
        if canJoin (i, j, delta) then
          begin
            delta : = (m [i, j] + delta) div 2;
            inc (ans, m [i, j]);
            used [j] : = true;
            for k : = 1 to n do
              if not used [k] and (k < > i) and (k < > j) then
                begin
                  dec (m [i, k], delta);
                  m [k, i] : = m [i, k];
                end;
            end;
          end;
        end;
      end;
    end;
  end;

procedure solve;
var
  c : integer;
begin
  prepare;
  for c : = n downto 2 do
    joinone;
  end;

procedure print;
begin
  writeln (ans);
end;

begin
  assign (input, inf); reset (input);
  assign (output, ouf); rewrite (output);
  while init do
    begin
      solve;
    end;
end;

```



```
print;
end;
close (output);
close (input);
end.
```

### 例题 6-6 树的中心问题。

问题描述:

给出一棵树, 求出树的中心。

为了定义树的中心, 首先给每个结点进行标号。对于一个结点  $K$ , 如果把  $K$  从树中删除 (连同与它相连的边一起), 剩下的被分成了很多块, 每一块显然又是一棵树 (即剩下的部分构成了一个森林)。则给结点  $K$  所标的号就是森林中结点个数最多的树所拥有的结点数。如果结点  $K$  的标号不大于其他任何一个结点的标号, 则结点  $K$  被称为是树的中心。

输入:

输入的第一行包含一个整数  $N$  ( $1 \leq N \leq 16\ 000$ ), 表示树中的结点数。接下来  $N-1$  行, 每个两个整数  $a, b$ , 由一个空格分隔, 表示  $a$  与  $b$  之间有一条边。

输出:

输出两行, 第一行两个整数  $v, T$ ,  $v$  表示树的中心结点的标号,  $T$  表示树有多少个中心。第二行包含  $T$  个数, 为所有树的中心的编号, 按升序排列。

样例输入:

```
7
1 2
2 3
2 4
1 5
5 6
6 7
```

样例输出:

```
3 1
1
```

分析:

和上题一样, 本题也是一个无根树问题, 需要规定一下为根。本题可以任意规定, 不妨规定为 1 号结点。

本题中, 对于一个结点, 删除这个结点后, 剩下的部分分为两类: 一类是原来这个结点的子树, 另一类是原来除这个结点和其子树以外的部分 (上方子树), 这些必定是一块, 如果知道一个结点所有的子树的结点数, 就可以得到除上方子树以外的所有块的结点数, 而:

上方子树的结点数 = 原来树的结点总数 ( $n$ ) - 所有子树的结点总数 - 1

所以, 这个问题只要求出每个结点的子结点总数即可。

源程序:

Const

maxn = 17000;

type

xtype = array [0..maxn] of integer;

var

n, min, tt : integer;

w : array [1..maxn] of ^xtype;

e : array [1..maxn, 1..2] of integer;

t : array [1..maxn] of integer;

pa : array [1..maxn] of integer;

top : integer;

stack : array [1..maxn] of integer;

data : array [1..maxn] of integer;

v : array [1..maxn] of integer;

procedure init;

var

i : integer;

begin

read (n);

for i := 1 to n - 1 do

begin

read (e [i, 1], e [i, 2]);

inc (t [e [i, 1]]);

inc (t [e [i, 2]]);

end;

end;

procedure prepare;

var

i : integer;

begin

for i := 1 to n do

begin

getmem (w [i], sizeof (integer) \* (t [i] + 1));

w [i] ^ [0] := 0;

end;

```

for i : = 1 to n - 1 do
begin
  inc (w [e [i, 1]] ^ [0]); w [e [i, 1]] ^ [w [e [i, 1]] ^ [0]] : = e [i, 2];
  inc (w [e [i, 2]] ^ [0]); w [e [i, 2]] ^ [w [e [i, 2]] ^ [0]] : = e [i, 1];
end;
end;

procedure solve;
begin
  fillchar (t, sizeof (t), 0);
  top : = 1;
  stack [top] : = 1;
  data [top] : = 0;
  while top > 0 do
  begin
    inc (data [top]);
    if data [top] > w [stack [top]] ^ [0] then
    begin
      if top > 1 then
      begin
        inc (t [stack [top - 1]], t [stack [top]]);
        if v [stack [top - 1]] < t [stack [top]] then
          v [stack [top - 1]] : = t [stack [top]];
        end;
        dec (top);
      end
    else
      if pa [stack [top]] < > w [stack [top]] ^ [data [top]] then
      begin
        stack [top + 1] : = w [stack [top]] ^ [data [top]];
        inc (top);
        data [top] : = 0;
        pa [stack [top]] : = stack [top - 1];
        t [stack [top]] : = 1;
      end;
    end;
  end;
end;

procedure printPrepare;

```



```

var
  i : integer;
begin
  min := n;
  tt := 1;
  for i := 1 to n do
  begin
    if n - t[i] > v[i] then v[i] := n - t[i];
    if v[i] < min then begin min := v[i]; tt := 1; end else
    if v[i] = min then inc(tt);
  end;
end;

procedure print;
var
  i : integer;
begin
  writeln(min, ' ', tt);
  for i := 1 to n do
    if v[i] = min then write(i, ' ');
  end;

begin
  init;
  prepare;
  solve;
  printPrepare;
  print;
end.

```

### 例题 6-7 笛卡尔树。

#### 问题描述：

让我们考虑一种特殊的二叉搜索树，叫笛卡尔树 (Cartesian Tree)。二叉搜索树是一棵有根的有序树，它对于它的任何一个结点  $x$ ，都满足以下条件：

它左子树中的每个结点的关键字都小于  $x$  的关键字，它右子树中的每个结点的关键字都大于  $x$  的关键字。也就是说，如果设  $x$  的左子树为  $L(x)$ ，右子树为  $R(x)$ ，关键字为  $K_x$ ，则，对于每个结点  $x$ ，有：

- \* 若  $y \in L(x)$  则  $K_y < K_x$
- \* 若  $z \in R(x)$  则  $K_z > K_x$

如果对二叉搜索树的每个结点  $x$  增加一个副关键字  $ax$ , 对于关键字  $ax$  满足堆的性质, 即:

若  $y$  是  $x$  的祖先结点, 则  $ay < ax$

则这棵二叉搜索树称为笛卡尔树。

给出系列的数对  $(k, a)$ , 用这些数对构造一棵笛卡尔树, 或者指出这样的二叉树是不可能构造出来的。

输入:

第一行包含一个整数  $N$ ——给出的数对的数目, 也就是笛卡尔树的结点数 ( $1 \leq N \leq 50000$ )。

接下来的  $N$  行每行包含一对数  $(K_i, a_i)$ , 所有的  $|K_i|, |a_i| \leq 30000$ 。所有的主关键字和副关键字都不同, 也就是: 对于任何  $i, j$  ( $i \neq j$ ), 有  $K_i \neq K_j$ , 且  $a_i \neq a_j$ 。

输出:

如果可以构造出对应的笛卡尔树, 在第一行输出 YES, 否则输出 NO。

如果有解, 输出对应的笛卡尔树。树的输出方法为:

首先将所有的结点按输入顺序从 1 至  $N$  编号。然后按顺序输出所有结点的信息, 每个结点的信息由三个数组成: 它的父结点, 左孩子结点, 右孩子结点。如果父结点或孩子结点不存在, 则输出 0。如果存在多于一种构树方案, 输出任意一棵即可。

样例输入:

```
7
5 4
2 2
3 9
0 5
1 3
6 6
4 11
```

样例输出:

```
YES
2 3 6
0 5 1
1 0 7
5 0 0
2 4 0
1 0 0
3 0 0
```

分析:

本题是一个建树的问题。如果将所有的关键字对按  $k$  关键字的升序排列, 则  $k$  关键字序列就是所要建的树的中序序列。考虑按顺序将所有的关键字对插入树中,  $now$  指针指向最后插入的结点。则  $now$  结点必然有以下性质:

1. 若  $now$  结点有父结点, 则  $now$  不是其父结点的左子树 (否则其父结点的中序遍历在  $now$  之后)。
2.  $now$  结点必然没有右子树 (否则其右子树的中序遍历在  $now$  之后)。

由于是按  $k$  关键字的升序插入, 因此, 再讨论  $k$  关键字已经无意义了, 我们只要在不改变以前节点的中序遍历先后顺序的情况下, 将要插入的节点插入在  $now$  的一个后继位置即可。

接下来考虑  $a$  关键字的问题。

设新插入的节点为  $node$ 。

情况 1:  $node$  的  $a$  关键字大于  $now$  的  $a$  关键字。

这种情况, 可以将  $node$  作为  $now$  的右子树,  $now$  作为  $node$  的父结点。如图 6-28。

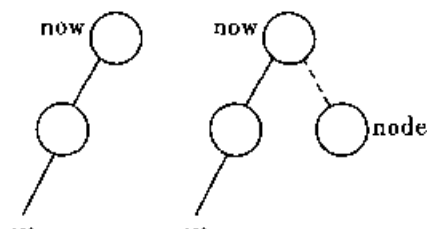


图 6-28 情况 1

情况 2:  $now$  没有父结点,  $node$  的  $a$  关键字小于  $now$  的  $a$  关键字。

这种情况, 可以将  $node$  作为  $now$  的父结点,  $now$  作为  $node$  的左子树。如图 6-29。

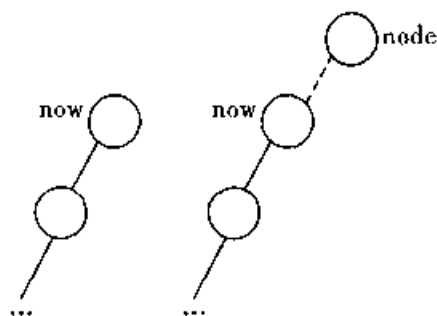


图 6-29 情况 2

情况 3:  $now$  有父结点  $par$ ,  $node$  的  $a$  关键字小于  $now$  的  $a$  关键字, 同时  $par$  的  $a$  关键字小于  $node$  的  $a$  关键字。

这种情况, 可以将  $node$  作为  $par$  的右子树, 然后将  $now$  作为  $node$  的左子树 (注意: 这里, 如果  $now$  有右子树, 也是一样可行的)。如图 6-30。

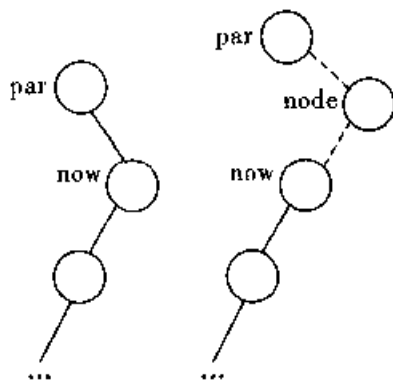


图 6-30 情况 3

情况 4:  $now$  有父结点  $par$ ,  $node$  的  $a$  关键字小于  $now$  的  $a$  关键字, 同时  $par$  的  $a$  关键字大于

node 的 a 关键字。

这种情况，可以将 now 设为 par，再插入 node 时不会影响到中序遍历的先后顺序。

源程序：

```
const
    maxr = 30001;
    maxn = 50001;

var
    n, now, r : longint;
    k, a : array [1..maxn] of integer;
    w : array [-maxr..maxr] of integer;
    ans : array [1..maxn, 1..3] of integer;

procedure init;
var
    i : integer;
begin
    read (n);
    for i := 1 to n do
    begin
        read (k[i], a[i]);
        w[k[i]] := i;
    end;
end;

procedure solve;
var
    i : integer;
begin
    for i := -maxr to maxr do
        if w[i] <> 0 then
        begin
            now := w[i];
            break;
        end;
    r := k[now] + 1;
    for i := r to maxr do
        if w[i] <> 0 then
        begin
```

```

    while (a[w[i]] < a[now]) and (ans[now, 1] < > 0) do
        now := ans[now, 1];
    if a[w[i]] < a[now] then
        begin
            ans[now, 1] := w[i];
            ans[w[i], 2] := now;
        end
    else
        begin
            ans[w[i], 2] := ans[now, 3];
            ans[now, 3] := w[i];
            ans[w[i], 1] := now;
            if ans[w[i], 2] < > 0 then
                ans[ans[w[i], 2], 1] := w[i];
            end;
            now := w[i];
        end;
    end;

procedure print;
var
    i : integer;
begin
    writeln('YES');
    for i := 1 to n do
        writeln(ans[i, 1], ' ', ans[i, 2], ' ', ans[i, 3]);
    end;

begin
    init;
    solve;
    print;
end.

```

### 例题 6-8 合并果子。

问题描述：

在一个果园里，多多已经将所有果子打了下来，而且按果子的不同种类分成了不同的堆。多多决定把所有的果子合成一堆。

每一次合并，多多可以把两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。可以

看出,所有的果子经过  $n-1$  次合并之后,就只剩下一堆了。多多在合并果子时总共消耗的体力等于每次合并所耗体力之和。

因为还要花大力气把这些果子搬回家,所以多多在合并果子时要尽可能地节省体力。假定每个果子重量都为 1,并且已知果子的种类数和每种果子的数目,你的任务是设计出合并的次序方案,使多多耗费的体力最少,并输出这个最小的体力耗费值。

例如有 3 种果子,数目依次为 1, 2, 9。可以先将 1、2 堆合并,新堆数目为 3,耗费体力为 3。接着,将新堆与原先的第三堆合并,又得到新的堆,数目为 12,耗费体力为 12。所以多多总共耗费体力  $= 3 + 12 = 15$ 。可以证明 15 为最小的体力耗费值。

输入:

输入文件 fruit.in 包括两行,第一行是一个整数  $n$  ( $1 \leq n \leq 10000$ ),表示果子的种类数。第二行包含  $n$  个整数,用空格分隔,第  $i$  个整数  $a_i$  ( $1 \leq a_i \leq 20000$ ) 是第  $i$  种果子的数目。

输出:

输出文件 fruit.out 包括一行,这一行只包含一个整数,也就是最小的体力耗费值。输入数据保证这个值小于 231。

样例输入:

3

1 2 9

样例输出:

15

数据规模:

对于 30% 的数据,保证有  $n \leq 1000$ ;

对于 50% 的数据,保证有  $n \leq 5000$ ;

对于全部的数据,保证有  $n \leq 10000$ 。

分析:

根据哈夫曼树带权路径长度与合并费用和之间的关系,本题最好的合并顺序应该用哈夫曼树的合并顺序,而所谓的最小体力耗费,也就是哈夫曼树的带权路径长度。

源程序:

```
const
  inf = 'fruit.in';
  outf = 'fruit.out';
  maxn = 30005;
var
  n, tot, ans : longint;
  a : array [1..maxn] of longint;

procedure readdata;
var
  i : longint;
begin
```



```

assign (input, inf);
reset (input);
readln (n);
for i : = 1 to n do
    read (a [i]);
close (input);
end;
procedure sort (l, r : longint);
var
    i, j : longint;
    x, y : longint;
begin
    i : = l; j : = r;
    x : = a [ (l + r) shr 1];
    repeat
        while a [i] < x do i : = i + 1;
        while a [j] > x do j : = j - 1;
        if i ≤ j then
            begin
                y : = a [i]; a [i] : = a [j]; a [j] : = y;
                i : = i + 1; j : = j - 1;
            end;
        until i > j;
        if l < j then sort (l, j);
        if i < r then sort (i, r);
    end;
procedure swap (i, j : longint);
var
    t : longint;
begin
    t : = a [i]; a [i] : = a [j]; a [j] : = t;
end;
procedure insertnode (x: longint);
var
    i, j : longint;
begin
    tot : = tot + 1;
    a [tot] : = x;
    i : = tot;

```



```

while i > 1 do
begin
  j := i shr 1;
  if a[i] < a[j] then
  begin
    swap(i, j);
    i := j;
  end
  else i := 1;
end;
end;
function deletemin : longint;
var
  i, j : longint;
begin
  deletemin := a[1];
  swap(1, tot);
  tot := tot - 1;
  i := 1;
  while i ≤ tot shr 1 do
  begin
    j := i + i;
    if (j < tot) and (a[j + 1] < a[j]) then j := j + 1;
    if a[j] < a[i] then
    begin
      swap(i, j);
      i := j;
    end
    else i := tot;
  end;
end;
procedure proceed;
var
  i, tmp : longint;
begin
  tot := n; ans := 0;
  for i := 1 to n - 1 do
  begin
    tmp := deletemin + deletemin;

```





```

    ans := ans + tmp;
    insertnode (tmp);
  end;
end;
procedure answer;
begin
  assign (output, out);
  rewrite (output);
  writeln (ans);
  close (output);
end;
begin
  readdata;
  sort (1, n);
  proceed;
  answer;
end.

```

### 例题 6-9 密码机。

#### 问题描述：

一台密码机按照以下方式产生密码：首先往机器中输入一系列数，然后取出其中一部分数，将它们异或以后得到一个新数作为密码。现在请模拟这样一台密码机的运行情况，用户通过输入控制命令来产生密码。

密码机中存放一个数列，初始时空。密码机的控制命令共有 3 种：

**ADD < Number >**

把 < Number > 加入到数列的最后。

**REMOVE < Number >**

在数列中找出一个等于 < Number > 的数，把它从数列中删除。

**XOR BETWEEN < Number1 > AND < Number2 >**

对于数列中所有大于等于 < Number1 > 并且小于等于 < Number2 > 的数依次进行异或，输出最后结果作为密码。如果只有一个数满足条件，输出这个数。如果没有任何满足条件，输出 0。

你可以假设用户不会 REMOVE 一个不存在于数列中的数，并且所有输入的数都不超过 20000。

**输入：**

输入文件 password.in 包括了一系列的控制命令。每一个控制命令占据单独一行。输入文件中没有多余的空行。文件不超过 60000 行。

**输出：**

对于每一个 XOR 命令，依次在 password.out 中输出一行包括你的密码所产生的密码。

输出文件中不应该包含任何的多余字符。

**输入样例：**

ADD 5

ADD 6

XOR BETWEEN 1 AND 10

REMOVE 5

XOR BETWEEN 1 AND 8

输出样例:

3

6

分析:

算法 1: 直接处理。

对于这样的试题, 最容易的方法就是直接处理。即: 设置一个数组  $Arr[1..t]$ , 用于存储数列中的所有元素, 设  $L$  为数组中的元素个数, 对于要求的三种运算, 分别处理如下:

ADD  $\langle number \rangle$ : 直接将  $\langle number \rangle$  加入数组的最末, 即  $L \leftarrow L + 1$ ,  $Arr[L] \leftarrow \langle number \rangle$ 。

REMOVE  $\langle number \rangle$ : 查找  $\langle number \rangle$  在数组中的位置, 设为  $p$ , 将  $p$  以后的所有元素前移一位即删除  $\langle number \rangle$ 。当然, 如果直接将  $Arr[p]$  值改为数组的最后一个元素值, 并删除最后一个元素, 可以少移动元素, 即:  $Arr[p] \leftarrow Arr[L]$ ,  $L \leftarrow L - 1$ 。

XOR BETWEEN  $\langle number1 \rangle$  AND  $\langle number2 \rangle$ : 设  $TmpResult = 0$ , 枚举数组中的每一个元素, 或其介于  $\langle number1 \rangle$  和  $\langle number2 \rangle$  (包括  $\langle number1 \rangle$  和  $\langle number2 \rangle$ ), 则将它与  $TmpResult$  取  $xor$  存于  $TmpResult$  中。最后  $TmpResult$  即为结果。

算法 1 采用最显而易见的方法处理问题。其正确性是毋庸置疑的。但是, 最直接的算法往往是很低效的。算法 1 的 REMOVE 和 XOR 操作的复杂度都为  $O(t)$ , 总的时间复杂度高达  $O(t^2)$ 。显然, 这对于题中如此大的数据范围是无法在规定时间内出解所有数据的。因此, 我们需要挖掘原题的性质, 以求更好地解决此题。

注意到其中的 XOR 操作, 都是对一个大小上的一段连续的数据操作, 因此, 可以考虑将所有的数保持按大小排好序。但是, XOR 操作, 如果直接计算, 必然会是  $O(t)$  的, 因此需要寻找更有效的方法。

考虑到:  $x1 \ xor \ x2 \ xor \ x3 = x1 \ xor \ (x2 \ xor \ x3)$ , 即  $xor$  操作满足结合律, 因此, 如果我们能既将整个序列分成很多段, 并用较低的复杂度更新每段的  $xor$  值, 就有可能更好地解决此问题。综合考虑, 线段树满足此要求。

算法 2: 采用线段树。

在算法 2 中, 我们采用线段树这种数据结构要处理每一种操作。首先, 令线段树  $T[a, b]$  表示数值大小在  $[a, b]$  这个区间内的信息, 包括左儿子指针  $Tz[a, b]$ 、右儿子指针  $Ty[a, b]$  以及处理 XOR 操作所需要的所有元素  $xor$  操作的结果。更新结点 (ADD, REMOVE) 时, 若  $T[a, b]$  是叶结点, 即  $a = b$ , 则可直接更新。若  $T[a, b]$  是分支结点, 则更新它的子结点, 显然只要更新左儿子结点或右儿子结点就行了。然后  $T[a, b] = Tz[a, b] \ xor \ Ty[a, b]$ 。

对于 XOR 操作, 我们只要找到  $[\langle number1 \rangle, \langle number2 \rangle]$  是哪几个区间的并, 然后将这些区间的值取  $xor$  即可。

算法 2 由于使用了线段树操作, 使得其每一种操作的时间复杂度都降为了  $O(\lg n)$ , 因此总和时间复杂度降为了  $O(t \lg n)$ , 已经是一种非常优秀的算法了。

但是,探索是永不停息的。下面,我们将通过继续分析,得到的一个时间复杂性的系数更低且更容易编程实现的算法,它仍然是基于一种优秀的数据结构的。

首先,我们提出一个疑问:是否有必要处理 REMOVE 操作?

考虑下面的式子:

$$x_2 \text{ xor } x_3 = x_1 \text{ xor } x_2 \text{ xor } x_3 \text{ xor } x_1$$

即,删除  $x_1$  后的数列的 xor 结果与再加一个  $x_1$  后的 xor 结果是完全一样的。这是由于 xor 运算不仅满足结合律,而满足交换律:  $x_1 \text{ xor } x_2 = x_2 \text{ xor } x_1$ , 且有一个非常特殊的性质:  $x \text{ xor } x = 0$ 。这样,对 xor 操作中删除一个数的操作可以用加上同一个数的操作来实现。这一条对于原题也同样适用。因此,REMOVE 操作可用 ADD 操作代替。

根据上面的分析,我们还可以得到这样一个式子:

$$(x_1 \text{ xor } x_2 \text{ xor } \cdots \text{ xor } x_n) \text{ xor } (x_1 \text{ xor } x_2 \text{ xor } \cdots \text{ xor } x_{a-1}) = (x_a \text{ xor } x_{j+1} \text{ xor } \cdots \text{ xor } x_b), \text{ 其中 } b \geq a.$$

即,对于区间  $[a, b]$  之间的 xor 值可通过  $[1, b]$  之间的 xor 值与  $[1, a]$  之间的 xor 值得到。这样,在记录时,就只要记录  $[1, x]$  这段的 xor 值即可。

算法 3: 树状数组。

基于上面的分析,我们可以找到一种更有效的数据结构:树状数组。通过维护  $[1, x]$  之间的 xor 值达到解决原题的目的。

这里,我们不再进行三种操作,而是进行两种操作:ADD 与 XOR, REMOVE 操作与 ADD 操作采用同一过程。

具体实现请见源程序。

源程序:

```
const
  inf = 'password. in';
  outf = 'password. out';
  maxn = 20000;

var
  v: array [1..maxn] of integer;
  ch: char;
  x: integer;
  i: integer;
  low, high: integer;

procedure Add (x: integer);
var
  p: word;
begin
  p := x;
```

```

while p ≤ maxn do
begin
  v [p] := v [p] xor x;
  p := (p or (p - 1)) + 1;
end;
end;

function gett (x: integer): integer;
var
  ans: integer;
begin
  ans := 0;
  while x > 0 do
  begin
    ans := ans xor v [x];
    x := x and (x - 1);
  end;
  gett := ans;
end;

begin
  assign (input, inf); reset (input);
  assign (output, outf); rewrite (output);
  while not seekeof do
  begin
    read (ch);
    if ch = 'A' then
    begin
      readln (ch, ch, x);
      Add (x);
    end
    else
    if ch = 'R' then
    begin
      readln (ch, ch, ch, ch, ch, x);
      Add (x);
    end
    else
    begin

```



```

    for i := 1 to 10 do read (ch);
    read (low);
    read (ch, ch, ch, ch);
    readln (high);
    if low > high then writeln (0) else writeln (gett (high) xor gett (low - 1));
  end;
end;
close (output);
close (input);
end.

```

### 例题 6-10 相等的单词。

#### 问题描述：

所有非空的 01 序列被称作一个二进制单词。一组相等的单词是如下形式的等式： $x_1x_2\cdots x_l = y_1y_2\cdots y_r$ ，这里  $x_i$  和  $y_j$  是二进制字符 01 或是用小写字母表示的变量。对每一个变量都有一个固定长度的两进制单词来代替这个变量，这个长度称之为这个变量的长度。为了解决单词相等的问题，我们需要用某种方法分配给所有变量适当的二进制单词（这个二进制单词的长度必须为这个变量的长度），使得变量被取代后的等式成立。对一个给定的等式计算有几种解答。

#### 例子：

让  $a, b, c, d, e$  分别为长度为 4, 2, 4, 4, 2 的 5 个变量。考虑以下等式： $1bad1 = acbe$ 。这个等式有 16 种不同的解答方案。

#### 任务：

写一个程序：

从文件 ROW.IN 中读入等式的数目以及它们的描述。

对每个等式找出它们的解答方案数。

将结果写入文件 ROW.OUT。

#### 输入：

在文件 ROW.IN 的第一行有一个整数  $x$  ( $1 \leq x \leq 5$ ) 表示等式的数目，随后有  $x$  个等式的描述。每个描述包括 6 行，两个等式的描述之间没有空行。每个等式用以下方式描述：在描述的第一行有一个整数  $k$  ( $0 \leq k \leq 26$ ) 表示等式中不同的变量数目，我们假设变量是从  $a$  起的  $k$  个小写字母。第二行有  $k$  个由空格隔开的正整数，表示  $k$  个变量的长度（第一个数表示  $a$  的长度，第二个数表示  $b$  的长度）。第三行有一个整数  $l$ ，表示等式左边的长度（有 0 1 及变量（单个字母）组成的单词长度）。等式左边将被写在下一行，仅包括 01 及小写字母而没有空格。以下两行给出了对等式左边的描述，第一行为一个正整数  $r$ ，表示等式右边的长度，等式的右边被写在第二行。等式两边所有变量的和相等且不超过 10000。

#### 输出：

对每个  $i$  ( $1 \leq i \leq x$ )，你的程序必须在第  $i$  行给出第  $i$  个等式的不同解答方案数，并将它写入文件 ROW.OUT。

输入样例:

```

1
5
4 2 4 4 2
5
1badl
4
acbe
    
```

输出样例:

```

16
    
```

分析:

这是一道关于含字母的等式的问题, 由于等式中变量的个数最多可以达到 10000, 因此不仅单纯的搜索不可能在短时间内出解, 就连  $O(n^2)$  的算法都很难奏效。

仔细观察这道题, 发现它实际上是字母对应的问题。如果仅仅只是每一个位置上的字母对应, 这道题就会变得很容易, 但事实上, 某个位置上的字母对应这样的单独的问题又被同一字母在等式中反复出现所联系起来, 因此问题就变得复杂起来了。那么首先让我们来研究一下这种对应之间的关系。

例如: 对于题目中的样例:

位置	1	2	3	4	5	6	7	8	9	10	11	12
等式左边	1	b1	b2	a1	a2	a3	a4	d1	d2	d3	d4	1
等式右边	a1	a2	a3	a4	c1	c2	c3	c4	b1	b2	e1	e2

每一列的对应并不是单独的, a1 出现了两次就把位置 1 与位置 4 联系了起来, 而 a4 又把位置 4 与位置 7 联系了起来, 1 又把位置 1 与位置 12 联系了起来, 所以我们得到位置 1, 4, 7, 12 是彼此相关联的, 又其中有 1 出现所以这四个位置上的数都是 1……

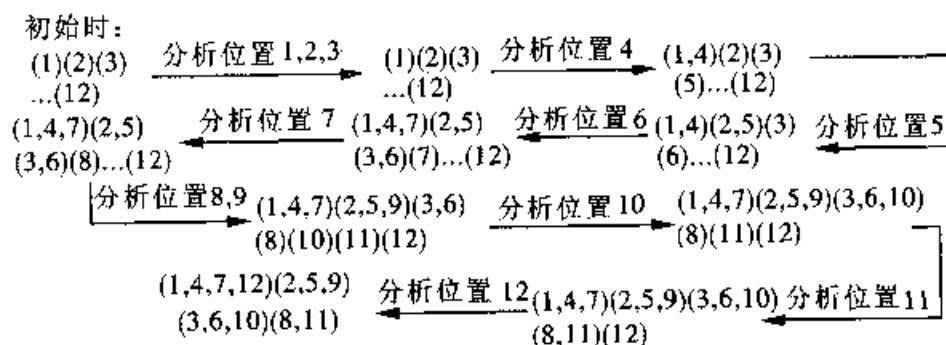
所以最终可以得到如下的位置组:

(1, 4, 7, 12), (2, 5, 9), (3, 6, 10), (8), (11)

同组之间的位置是彼此相关的, 决定了其中任意一个位置上的取值, 其他位置也就随之而确定了。而两组之间又是彼此相互独立的。去掉第一个位置组已经确定是 1, 共有 4 个可以随机取值的位置组, 所以答案是  $2^4 = 16$ 。

通过上面的例子我们可以看到: 这道题实际上就是求这样两个 01 串中有几个位置是可以互不相关的随机取值的, 或者说就是确定有几个如上例中的位置组。而每一个位置上的对应关系就可以看成是位置组之间的桥梁, 或者可以把当前的位置组信息看作是一些集合, 则新的一个位置上的对应关系就可以看成是对集合元素的合并, 到这里问题的答案已经很明显了, 我们可以用并查集来实现。注意: 如果 0 与 1 被放入了同一个集合则失败退出。

比如对于上面的样例:



然而由于这样的方法在每个位置上要确定其字母在其他地方是否出现过,实现比较困难,我们可以稍作修改使得实现上容易一些。

将所有的字母每一位上(字母代表多个数字)所在集合编号放入一个数组  $s$  中,并用  $s[0]$  表示数字 0 的所在集合编号,  $s[1]$  表示数字 1 的。举个例子说,对  $a$ 、 $b$ 、 $c$ 、 $d$ 、 $e$  长度分别为 4、2、4、4、2 时,  $s[2]$  表示  $a[1]$  的,  $s[3]$  表示  $a[2]$  的,  $s[6]$  表示  $b[1]$  的,而  $s[16]$  表示  $e[1]$  的。

再来看刚才的样例:

数组  $s$  (下标):

0 1 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17

数组  $s$  (代表):

0 1 a1 a2 a3 a4 b1 b2 c1 c2 c3 c4 d1 d2 d3 d4 e1 e2

初始:

0 1 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17

分析位置 1:

0 1 01 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17

分析位置 2:

0 1 01 03 04 05 03 07 08 09 10 11 12 13 14 15 16 17

分析位置 3:

0 1 01 03 04 05 03 04 08 09 10 11 12 13 14 15 16 17

分析位置 4:

0 1 01 03 04 01 03 07 08 09 10 11 12 13 14 15 16 17

分析位置 5:

0 1 01 03 04 01 03 07 03 09 10 11 12 13 14 15 16 17



分析位置 6:

0 1 01 03 04 01 03 07 03 04 10 11 12 13 14 15 16 17

分析位置 7:

0 1 01 03 04 01 03 07 03 04 05 11 12 13 14 15 16 17

分析位置 8:

0 1 01 03 04 01 03 07 03 04 05 11 11 13 14 15 16 17

分析位置 9:

0 1 01 03 04 01 03 07 03 04 05 11 11 03 14 15 16 17

分析位置 10:

0 1 01 03 04 01 03 07 03 04 05 11 11 03 04 15 16 17

分析位置 11:

0 1 01 03 04 01 03 07 03 04 05 11 11 03 04 15 15 17

分析位置 12:

0 1 01 03 04 01 03 07 03 04 05 11 11 03 04 15 15 01

最终检查一下所在集合编号仍为自己本身的集合个数, 注意要除去  $s[0]$  与  $s[1]$ 。

关于并查集, 利用将小的树合并到大的树上去以及路径压缩后, 可以基本上在  $O(n)$  的时间内出解。

时间复杂度:  $O(na(n))$ ;  $a(n)$  随  $n$  增长极为缓慢, 可以认为是一个常数。

空间复杂度:  $O(n)$ 。

源程序:

```
program row;
const maxn = 10000;
var s : array [0..2*maxn] of integer; {用于并查集}
    long : array [0..25, 1..2] of integer;
    {存储字母在数组 s 中的起始位置以及字母代表的 01 串长度}
    t : array [1..maxn] of integer; {存储等式左边}
    num, ii : integer;
    v : array [0..25] of boolean;
    {记录等式中出现过的字母}
procedure make; {主过程}
var i, nn, ans, n, j, tl : integer;
procedure readin; {读入过程}
var i, j, k, now, m : integer;
```





```

x                                     : char;
begin
  fillchar (long, sizeof (long), 0);
  fillchar (v, sizeof (v), false);
  readln (n);
  | 读入并计算数组 long |
  for i: =0 to n-1 do read (long [i, 2]);
  long [0, 1]: =2;
  for i: =1 to n do long [i, 1]: =long [i-1, 1] + long [i-1, 2];
  nn: =long [n, 1] -1;
  | 读入数组 t 以及计算数组 v |
  fillchar (t, sizeof (t), 0);
  readln (m); i: =0;
  for j: =1 to m do
  begin
    read (x);
    if (x = '1') or (x = '0') then
    begin
      inc (l);
      t [i]: =ord (x) - ord ('0');
    end
    else
    begin
      now: =ord (x) - ord ('a'); inc (l);
      v [now]: =true;
      for k: =i to i+long [now, 2] -1 do
      begin
        t [k]: =long [now, 1] + k - i;
      end;
      inc (i, long [now, 2] -1);
    end;
  end;
  tl: =i;
end;
function fa (x                       : integer): integer;
| 返回 x 所在树的根结点并进行路径压缩 |
var i, j, k                           : integer;
begin
  i: =x;

```



```

while s [ i ] < > i do i: = s [ i ];
fa: = i; j: = i;
i: = x;
while s [ i ] < > j do begin k: = s [ i ]; s [ i ]: = j; i: = k; end;
end;
procedure add ( x, y          : integer ); { 合并 x, y }
var x1, x2                    : integer;
begin
    x1: = fa ( x ); x2: = fa ( y );
    if x1 < > x2 then s [ x2 ]: = x1;
end;
procedure run;
{ 读入等式右边并与数组 t 比较进行集合合并操作 }
var x                          : char;
    i, j, m, now, k            : integer;
begin
    readln ( m ); i: = 0;
    for j: = 1 to m do
        begin
            read ( x ); { 读入当前元素 }
            if ( x = '1' ) or ( x = '0' ) then
                begin
                    inc ( i );
                    add ( ord ( x ) - ord ( '0' ), t [ i ] );
                end
            else
                begin
                    now: = ord ( x ) - ord ( 'a' ); inc ( i );
                    v [ now ]: = true;
                    for k: = i to i + long [ now, 2 ] - 1 do { 集合合并 }
                        begin
                            add ( long [ now, 1 ] + k - i, t [ k ] );
                        end;
                    inc ( i, long [ now, 2 ] - 1 );
                end;
        end;
    end;
    if i < > tl then s [ 1 ]: = s [ 0 ]; { 如果等式两边长度不同, 失败退出 }
end;
procedure writeout ( ans: integer );

```



```

}输出过程: 输出 2*ans{
var a : array [1..10000] of integer;
    aa, i, now : integer;
procedure cheng; {高精度乘法}
var x, y : integer;
    i : integer;
begin
    x := 0;
    for i := 1 to aa do
        begin
            y := (a[i] * 1024 + x) div 10;
            a[i] := (a[i] * 1024 + x) mod 10;
            x := y;
        end;
    while x < > 0 do begin inc(aa); a[aa] := x mod 10; x := x div 10; end;
end;
begin
    if ans = 0 then begin writeln('1'); exit; end;
    fillchar(a, sizeof(a), 0); aa := 0; now := 1;
    for i := 1 to (ans mod 10) do
        now := now * 2;
    while now < > 0 do begin inc(aa); a[aa] := now mod 10; now := now div 10; end;
    for i := 1 to (ans div 10) do
        cheng;
    for i := aa downto 1 do
        write(a[i]);
    writeln;
end; {outans}
begin {make}
    readln;
    for i := 0 to nn do s[i] := i;
    run;
    if fa(0) = fa(1) then begin writeln('0'); exit; end;
    {若 01 属于一个位置组, 则失败退出}
    ans := 0;
    {若某个字母未出现, 则将其父结点标记为 0}
    for i := 0 to n - 1 do
        if not v[i] then
            begin

```



```

for j: =long [i, 1] to long [i, 1] +long [i, 2] -1 do
  s [j]: =0;
end;
for i: =2 to nn do if s [i] =i then inc (ans); |统计根结点的个数|
if fa (0) <>0 then dec (ans); |除去 01|
if fa (1) <>1 then dec (ans);
writeout (ans);
end; |main procedure|
begin
  assign (input, 'row.in'); reset (input);
  assign (output, 'row.out'); rewrite (output);
  readln (num);
  for ii: =1 to num do
    make;
  close (input); close (output);
end.

```

## 6.12 小 结

树是一种非常重要的数据结构。它的用途非常广泛，很多算法都直接或间接地用树来做，很多题目都能向树转换。更重要的，它并不只是一种单纯的数据结构，其一对多的结构、递归的思想在信息学及很多地方都是值得借鉴的。

## 习题六

### 一、单选题

1. 树中所有结点的度等于所有结点数加( )。  
A. 0                      B. 1                      C. -1                      D. 2
2. 在一棵二叉树的二叉链表中，空指针域数等于非空指针域数加( )。  
A. 2                      B. 1                      C. 0                      D. -1
3. 在一棵具有  $n$  个结点的二叉树中，所有结点的空子树个数等于( )。  
A.  $n$                       B.  $n-1$                       C.  $n+1$                       D.  $2n$
4. 在一棵具有  $n$  个结点的二叉树的第  $i$  层上，最多具有( )个结点。  
A.  $2i$                       B.  $2^{i-1}$                       C.  $2^{i-1}$                       D.  $2^n$
5. 在一棵具有 35 个结点的完全二叉树中，该树的深度为( )。  
A. 6                      B. 7                      C. 5                      D. 8

6. 在一棵具有  $n$  个结点的完全二叉树中, 树枝结点的最大编号为( )。
- A.  $\lceil (n+1)/2 \rceil$                       B.  $\lfloor (n+1)/2 \rfloor$   
 C.  $\lceil n/2 \rceil$                       D.  $\lfloor n/2 \rfloor$
7. 在一棵完全二叉树中, 若编号为  $i$  的结点存在左孩子, 则左孩子结点的编号为( )。
- A.  $2i$                       B.  $2i-1$                       C.  $2i+1$                       D.  $2i+2$
8. 在一棵完全二叉树中, 对于编号为  $i$  ( $i>1$ ) 的结点, 其双亲结点的编号为( )。
- A.  $\lceil (i-1)/2 \rceil$                       B.  $\lfloor (i-1)/2 \rfloor$   
 C.  $\lceil i/2 \rceil$                       D.  $\lfloor i/2 \rfloor$
9. 一棵二叉树的广义表表示为  $a(b, c, d(e, g(h)), f))$ , 则该二叉树的高度为( )。
- A. 3                      B. 4                      C. 5                      D. 6
10. 从二叉搜索树中查找一个元素时, 其时间复杂度大致为( )。
- A.  $O(n)$                       B.  $O(1)$                       C.  $O(\log_2 n)$                       D.  $O(n^2)$
11. 根据  $n$  个元素建立一棵二叉搜索树时, 其时间复杂度大致为( )。
- A.  $O(n)$                       B.  $O(\log_2 n)$                       C.  $O(n^2)$                       D.  $O(n \log_2 n)$
12. 从堆中删除一个元素的时间复杂度为( )。
- A.  $O(1)$                       B.  $O(n)$                       C.  $O(\log_2 n)$                       D.  $O(n \log_2 n)$

## 二、填空题

1. 对于一棵具有  $n$  个结点的树, 该树中所有结点的度数之和为\_\_\_\_\_。
2. 假定一棵三叉树的结点个数为 50, 则它的最小深度为\_\_\_\_\_, 最大深度为\_\_\_\_\_。
3. 在一棵高度为  $h$  的四叉树中, 最多含有\_\_\_\_\_个结点。
4. 一棵深度为 5 的满二叉树中的结点数为\_\_\_\_\_个, 一棵深度为 3 的满四叉树中的结点数为\_\_\_\_\_个。
5. 在一棵二叉树中, 假定双分支结点数为 5 个, 单分支结点数为 6 个, 则叶子结点数为\_\_\_\_\_个。
6. 一棵二叉树的广义表表示为  $a(b(c, d), e(f(, g)))$ , 它含有双分支结点\_\_\_\_\_个, 单分支结点\_\_\_\_\_个, 叶子结点\_\_\_\_\_个。
7. 对于一棵含有 40 个结点的理想平衡树, 它的高度为\_\_\_\_\_。
8. 若对一棵二叉树从 0 开始进行结点编号, 并按此编号把它顺序存储到一维数组  $a$  中, 即编号为 0 的结点存储到  $a[0]$  中, 其余类推, 则  $a[i]$  元素的左孩子元素为\_\_\_\_\_, 右孩子元素为\_\_\_\_\_, 双亲元素 ( $i>0$ ) 为\_\_\_\_\_。
9. 对于一棵具有  $n$  个结点的二叉树, 对应二叉链表中指针总数为\_\_\_\_\_个, 其中\_\_\_\_\_个用于指向孩子结点, \_\_\_\_\_个指针空闲着。
10. 在一棵高度为 5 的理想平衡树中, 最少含有\_\_\_\_\_个结点, 最多含有\_\_\_\_\_个结点。
11. 对一棵二叉搜索树进行中序遍历时, 得到的结点序列是一个\_\_\_\_\_。
12. 当向一个小根堆插入一个具有最小值的元素时, 该元素需要逐层\_\_\_\_\_调整, 直到被调整到\_\_\_\_\_位置为止。

### 三、运算题

1. 假定一棵二叉树广义表表示为  $a(b, c, d(e, f))$ ，分别写出对它进行先序、中序、后序、按层遍历的结果。

先序：

中序：

后序：

按层：

2. 已知一棵二叉树的先根和中根序列，求该二叉树的后根序列。

先根序列：A, B, C, D, E, F, G, H, I, J

中根序列：C, B, A, E, F, D, I, H, J, G

后根序列：

3. 已知一棵二叉树的中根和后根序列，求该二叉树的高度和双支、单支及叶子结点数。

中根序列：c, b, d, e, a, g, i, h, j, f

后根序列：c, e, d, b, i, j, h, g, f, a

高度：      双支：      单支：      叶子：

4. 已知一组元素为 (46, 25, 78, 62, 12, 37, 70, 29)，画出按元素排列顺序输入生成的一棵二叉搜索树，再以广义表形式给出该二叉搜索树。

5. 从空堆开始依次向小根堆中插入集合 {38, 64, 52, 15, 73, 40, 48, 55, 26, 12} 中的每个元素，请以顺序表的形式给出每插入一个元素后堆的状态。

6. 已知一个堆为 (12, 15, 40, 38, 26, 52, 48, 64)，若需要从堆中依次删除四个元素，请给出每删除一个元素后堆的状态。

### 四、上机编程题

#### 1. 促销。

题目描述：

Bytelandish 超市营销部要你为其编写一程序来模拟计算正在准备的促销活动的费用。

此次促销活动遵循以下规则：

(1) 参与促销活动的消费者应在其付款的账单上填写个人的详细信息，并将账单放入一特殊的投票箱内；

(2) 每一促销日的最后，从投票箱中取出两张账单：数额最大的和数额最小的；支付最大账单的顾客将会得到数值等于两账单之差的现金奖励；

(3) 为了防止一次购买却获得多次奖励的现象发生，每次抽出的两张账单都不再放回投票箱，但箱内的剩余账单继续参与此次促销活动。

由于超市的营业额非常大，因而我们可以认为在每个促销日的最后，在我们取出最大与最小账单之前，投票箱内至少有两张账单。

任务：

编写一程序

(1) 促销活动中每天放入投票箱内的账单价格表从文件 PRO. IN 中读入；

(2) 计算在这连续的促销活动中用做奖励的总费用；

(3) 结果写入文件 PRO. OUT。



输入:

输入文件 PRO. IN 的第一行的正整数  $N$  表示为此次促销活动持续的天数,  $1 \leq N \leq 5000$ 。接下来的  $N$  行含有一连串用单个空格隔开的非负数。第  $1+i$  行的数字表示在第  $i$  天促销活动中放入投票箱内账单的价格。此行的第一个整数  $K$  表示这一天放入投票箱内的账单数,  $0 \leq K \leq 10^5$ , 接着的  $K$  个正整数分别代表着每张账单的价格, 所有的这些数字都不大于 10 的 6 次幂。

整个活动中放入投票箱内的账单数不超过 10 的 6 次幂。

输出:

输出文件 PRO. OUT 中恰含有一整数, 它的值等于此次促销活动用作奖励的总花费。

输入输出样例:

输入:	输出:
5	19
3 1 2 3	
2 1 1	
4 10 5 5 1	
0	
1 2	

## 2. 商务旅行。

题目描述:

某首都城市的商人要经常到各城镇去做生意, 他们按自己的路线去做, 目的是为了节约时间。

假设有  $N$  个城镇, 首都编号为 1, 商人从首都出发, 其他各城镇之间都有道路连接, 任意两个城镇之间如果有直连道路, 在他们之间行驶需要花费单位时间。该国公路网络发达, 从首都出发能到达任意一个城镇, 并且公路网络不会存在环。

你的任务是帮助该商人计算一下他的最短旅行时间。

输入描述:

输入文件 kom. in 中的第一行有一个整数  $N$ ,  $1 \leq n \leq 30\,000$ , 为城镇的数目。下面  $N-1$  行, 每行由两个整数  $a$  和  $b$  ( $1 \leq a, b \leq n; a < b$ ) 组成, 表示城镇  $a$  和城镇  $b$  有公路连接。在第  $N+1$  行为一个整数  $M$  ( $2 \leq M \leq 10^6$ ), 下面的  $M$  行, 每行有该商人需要顺次经过的各城镇编号。

输出描述:

在输出文件 kom. out 中输出该商人旅行的最短时间。

输入输出示例:

输入:	输出:
5	7
1 2	
1 5	
3 5	
4 5	
4	
1	



3  
2  
5

### 3. 区间染色判定问题。

题目描述：

有  $n$  个区间，其起点和终点分别用  $a_i$  和  $b_i$  表示。对于两个区间  $i$  和  $j$ ，若  $a_i < a_j < b_i < b_j$  或  $a_j < a_i < b_j < b_i$  则称区间  $i$  和区间  $j$  相交。对所有的区间染黑白二色，每个区间至少要染其中的一种颜色。假设两个区间同色且相交，那么就称这种染色方案不合法。给定  $n$  个区间和一个染色方案，问这个方案是否合法？

输入描述：

输入文件第一行是区间数  $n$ ；

此后  $n$  行，每行三个数  $a_i, b_i, c_i$ ，分别表示区间的起点和终点以及区间的颜色（ $c_i = 0$  是白色， $c_i = 1$  是黑色）。

输出描述：

若这个方案合法则输出 “Accept”，否则输出 “WA”。

数据范围：

$1 \leq n \leq 100000$ ；

$1 \leq a_i \leq b_i \leq 1000000000$ 。

### 4. 银河英雄传说。

题目描述：

公元 5801 年，地球居民迁移至金牛座  $\alpha$  第二行星，在那里发表银河联邦创立宣言，同年改元为宇宙历元年，并开始向银河系深处拓展。

宇宙历 799 年，银河系的两大军事集团在巴米利恩星域爆发战争。泰山压顶集团派宇宙舰队司令莱因哈特率领十万余艘战舰出征，气吞山河集团点名将杨威利组织麾下三万艘战舰迎敌。

杨威利擅长排兵布阵，巧妙运用各种战术屡次以少胜多，难免滋生骄气。在这次决战中，他将巴米利恩星域战场划分成 30000 列，每列依次编号为 1, 2, ..., 30000。之后，他把自己的战舰也依次编号为 1, 2, ..., 30000，让第  $i$  号战舰处于第  $i$  列（ $i = 1, 2, \dots, 30000$ ），形成“一字长蛇阵”，诱敌深入。这是初始阵形。当进犯之敌到达时，杨威利会多次发布合并指令，将大部分战舰集中在某几列上，实施密集攻击。合并指令为  $Mij$ ，含义为让第  $i$  号战舰所在的整个战舰队列，作为一个整体（头在前尾在后）接至第  $j$  号战舰所在的战舰队列的尾部。显然战舰队列是由处于同一列的一个或多个战舰组成的。合并指令的执行结果会使队列增大。

然而，老谋深算的莱因哈特早已在战略上取得了主动。在交战中，他可以通过庞大的情报网络随时监听杨威利的舰队调动指令。

在杨威利发布指令调动舰队的同时，莱因哈特为了及时了解当前杨威利的战舰分布情况，也会发出一些询问指令： $Cij$ 。该指令意思是，询问电脑，杨威利的第  $i$  号战舰与第  $j$  号战舰当前是否在同一列中；如果在同一列中，那么它们之间布置有多少战舰。

作为一个资深的高级程序设计员，你被要求编写程序分析杨威利的指令，以及回答莱因哈特的询问。

最终的决战已经展开，银河的历史又翻过了一页……





输入描述:

输入文件 galaxy.in 的第一行有一个整数  $T$  ( $1 \leq T \leq 500000$ ), 表示总共有  $T$  条指令。

以下有  $T$  行, 每行有一条指令。指令有两种格式:

$M\ i\ j$ :  $i$  和  $j$  是两个整数 ( $1 \leq i, j \leq 30000$ ), 表示指令涉及的战舰编号。该指令是莱因哈特窃听到的杨威利发布的舰队调动指令, 并且保证第  $i$  号战舰与第  $j$  号战舰不在同一列。

$C\ i\ j$ :  $i$  和  $j$  是两个整数 ( $1 \leq i, j \leq 30000$ ), 表示指令涉及的战舰编号。该指令是莱因哈特发布的询问指令。

输出描述:

输出文件为 galaxy.out。你的程序应当依次对输入的每一条指令进行分析和处理:

如果是杨威利发布的舰队调动指令, 则表示舰队排列发生了变化, 你的程序要注意到这一点, 但是不要输出任何信息。

如果是莱因哈特发布的询问指令, 你的程序要输出一行, 仅包含一个整数, 表示在同一列上, 第  $i$  号战舰与第  $j$  号战舰之间布置的战舰数目。如果第  $i$  号战舰与第  $j$  号战舰当前不在同一列上, 则输出  $-1$ 。

输入输出样例:

输入:	输出:
4	-1
M 2 3	1
C 1 2	
M 2 4	
C 4 2	

样例说明:

战舰位置图: 表格中阿拉伯数字表示战舰编号

	第一列	第二列	第三列	第四列	.....
初始时	1	2	3	4	.....
M 2 3	1		3 2	4	.....
C 1 2	1 号战舰与 2 号战舰不在同一列, 因此输出 -1				
M 2 4	1			4 3 2	.....
C 4 2	4 号战舰与 2 号战舰之间仅布置了一艘战舰, 编号为 3, 输出 1				

## 7 图

### 7.1 图的概念

图 (graph) 是图型结构的简称。它是一种复杂的非线性数据结构。图在各个领域都有着广泛的应用。图的二元组定义为:

$$G = (V, E)$$

其中  $V$  是非空的顶点集合, 即

$$V = \{v_i \mid 1 \leq i \leq n, n \geq 1, v_i \in \text{elementype}, n \text{ 为顶点数}\}$$

$E$  是  $V$  上二元关系的集合, 一般我们只讨论仅含一个二元关系的情况, 且直接用  $E$  表示这个关系。这样,  $E$  就是  $V$  上顶点的序偶或无序对 (每个无序对  $(x, y)$  是两个对称序偶  $\langle x, y \rangle$  和  $\langle y, x \rangle$  的简写形式) 的集合。对于  $V$  上的每个顶点, 在  $E$  中都允许有任意多个前驱和任意多个后继, 即对每个顶点的前驱和后继个数均不限制。回顾一下线性表和树的二元组定义, 都是在其二元关系上规定了某种限制, 线性表的限制是只允许每个结点有一个前驱和一个后继, 树的限制是只允许每个结点有一个前驱和多个后继。因此, 图比线性表和树具有更广泛性, 它包含线性表和树在内, 线性表和树可看作图的简单情况。

对于一个图  $G$ , 若  $E$  是序偶 (有序对) 的集合, 则每个序偶对应图形中的一条有向边; 若  $E$  是无序对的集合, 则每个无序对对应图形中的一条无向边, 所以可把  $E$  看作是边的集合。这样图的二元组定义可叙述为: 图由非空顶点集 (vertexset) 和边集 (edgeset) 所组成。针对图  $G$ , 顶点集和边集可分别记为  $V(G)$  和  $E(G)$ 。边集  $E(G)$  允许是空集, 这时图  $G$  种的顶点均为孤立顶点。

对于一个图  $G$ , 若边集  $E(G)$  为有向边, 则称此图为有向图 (digraph), 若边集  $E(G)$  为无向边, 则称此图为无向图 (undigraph)。如图 7-1 中所示的  $G_1$  是无向图,  $G_2$  是有向图。

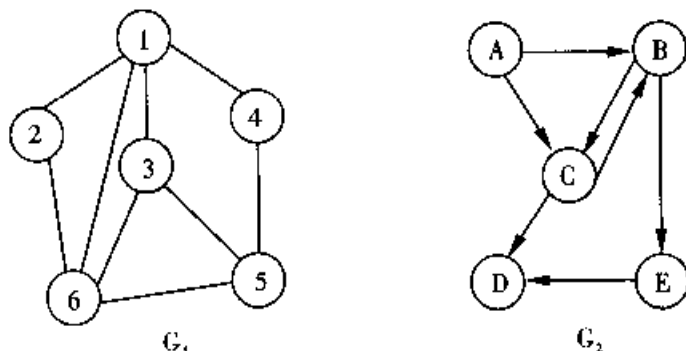


图 7-1 图的示例

## 7.2 图的基本术语

### 1. 端点和邻接点

在一个无向图中,若存在一条边  $(v_i, v_j)$ ,则称  $v_i, v_j$  为此边的两个端点,并称它们互为邻接点 (adjacent),即  $v_i$  是  $v_j$  的一个邻接点,  $v_j$  也是  $v_i$  的一个邻接点。如图 7-1 的  $G_1$  中,以顶点  $v_1$  的四条边是  $(1, 2), (1, 3), (1, 4)$  和  $(1, 6)$ ,  $v_1$  四个邻接点分别为  $v_2, v_3, v_4$  和  $v_6$ 。

在一个有向图中,若存在一条边  $\langle v_i, v_j \rangle$ ,则称此边是顶点  $v_i$  的一条出边 (outedge),顶点  $v_j$  的一条入边 (inedge);称  $v_i$  为此边的起始端点,简称起点或始点,  $v_j$  为此边的终止端点,简称终点;称  $v_i$  和  $v_j$  互为邻接点,并称  $v_j$  是  $v_i$  的出边邻接点,  $v_i$  是  $v_j$  的入边邻接点。如图 7-1 的  $G_2$  中,顶点 C 有两条出边  $\langle C, B \rangle$  和  $\langle C, D \rangle$ ,两条入边  $\langle A, C \rangle$  和  $\langle B, C \rangle$ ,顶点 C 的两个出边邻接点为  $v_B$  和  $v_D$ ,两个入边邻接点为  $v_A$  和  $v_B$ 。

### 2. 顶点的度、入度、出度

无向图顶点  $v$  的度 (degree) 定义为以该顶点为一个端点的边的数目,简单地说,就是该顶点的边的数目,记为  $D(v)$ 。如图 7-1 的  $G_1$  中  $v_1$  顶点的度为 4,  $v_2$  顶点的度为 2。有向图中顶点  $v$  的度有入度和出度之分,入度 (indegree) 是该顶点的入边的数目,记为  $ID(v)$ ;出度 (outdegree) 是该顶点的出边的数目,记为  $OD(v)$ 。顶点  $v$  的度等于它的入度和出度之和,即  $D(v) = ID(v) + OD(v)$ 。如图 7-1 的  $G_2$  中顶点 A 的入度为 0,出度为 2,度为 2;顶点 C 的入度为 2,出度为 2,度为 4。

若一个图中有  $n$  个顶点和  $e$  条边,则该图所有顶点的度同边数  $e$  满足下面关系:

$$e = \frac{1}{2} \sum_{i=1}^n D(v_i)$$

这很容易理解,因为每条边的度数为 2,所以全部顶点的度数为所有边数的 2 倍,或者说,边数为全部顶点的度数的一半。

### 3. 完全图、稠密图、稀疏图

若无向图中的每两个顶点之间都存在一条边,有向图中的每两个顶点之间都存在着方向相反的两条边,则称此图为完全图。显然,若完全图是无向的,则图中包含有  $n(n-1)/2$  条边;

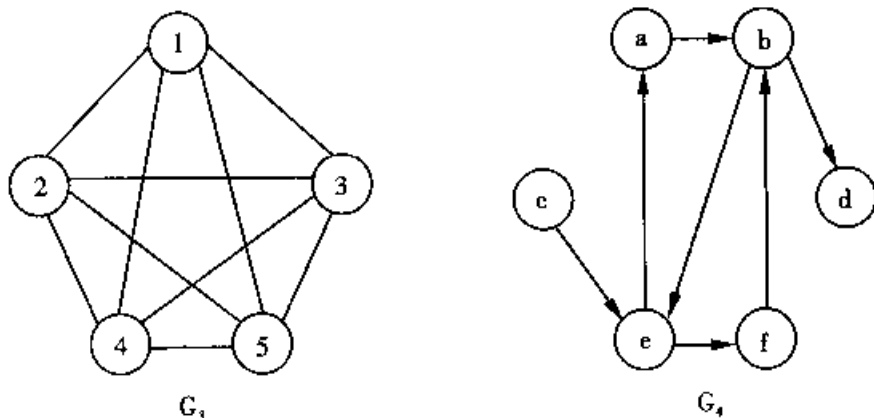


图 7-2 完全图和稀疏图示例

若完全图是有向的, 则图中包含有  $n(n-1)$  条边。当一个图接近完全图时, 则可称为稠密图, 相反地, 当一个图有较少的边数 (即  $e \ll n(n-1)$ ) 时, 则可称为稀疏图。图 7-2 中的  $G_3$  就是一个具有五个顶点的无向完全图,  $G_4$  就是一个具有六个顶点的稀疏图。

#### 4. 子图

设有两个图  $G = (V, E)$  和  $G' = (V', E')$ , 若  $V'$  是  $V$  的子集, 且  $E'$  是  $E$  的子集, 则称  $G'$  是  $G$  的子图。例如图 7-2, 由  $G_3$  中全部顶点和同  $v_1$  相邻的所有边可构成  $G_3$  的一个子图, 由  $G_3$  中的顶点  $v_1, v_2, v_3$  和他们之间的所有边可构成  $G_3$  的另一个子图。

#### 5. 路径和回路

在一个图  $G$  中, 从顶点  $v$  到顶点  $v'$  的一条路径 (path) 是一个顶点序列  $v_0, v_1, v_2, \dots, v_m$ , 其中  $v = v_0, v' = v_m$ , 若此图是无向图, 则  $(v_{j-1}, v_j) \in E(G) (1 \leq j \leq m)$ ; 若此图是有向图, 则  $\langle v_{j-1}, v_j \rangle \in E(G) (1 \leq j \leq m)$ 。路径长度是指该路径上经过的边的数目。若一条路径上除了前后端点可以相同外, 所有顶点均不同, 则称此路径为简单路径。若一条路径上的前后两端点相同, 则被称为回路或环 (cycle), 前后两端点相同的简单路径被称为简单回路或简单环。如图 7-2 的  $G_4$  中, 从顶点  $c$  到顶点  $d$  的一条路径为  $v_c, v_e, v_a, v_b, v_d$ , 其路径长度为 4; 路径  $v_a, v_b, v_c, v_a$  为一条简单回路, 其路径长度为 3; 路径  $v_a, v_b, v_c, v_f, v_b$  不是一条简单路径, 因为存在着从顶点  $v_b$  到  $v_b$  的一条回路。

#### 6. 连通和连通分量

在无向图  $G$  中, 若从顶点  $v_i$  到顶点  $v_j$  有路径, 则称  $v_i$  和  $v_j$  是连通的。若图  $G$  中任意两个顶点都连通, 则称  $G$  为连通图, 否则称为非连通图。无向图  $G$  的极大连通子图称为  $G$  的连通分量。显然, 任何连通图的连通分量只有一个, 即本身, 而非连通图有多个连通分量。例如, 上面给出的图 7-1 中的  $G_1$  和图 7-2 中的  $G_3$  就是连通图。

#### 7. 强连通图和强连通分量

在有向图  $G$  中, 若从顶点  $v_i$  到顶点  $v_j$  有路径, 则称从  $v_i$  到  $v_j$  是连通的。若图  $G$  中的任意两个顶点  $v_i$  和  $v_j$  都连通, 即从  $v_i$  到  $v_j$  和从  $v_j$  到  $v_i$  都存在路径, 则称  $G$  是强连通图。有向图  $G$  的极大强连通子图称为  $G$  的强连通分量。显然, 强连通图只有一个强连通分量, 即本身, 非强连通图有多个强连通分量。

#### 8. 权和网

在一个图中, 每条边可以标上具有某种含义的数值, 此数值称为该边的权 (weight)。例如, 对于一个反映城市交通线路的图, 边上的权可表示该条线路的长度或等级; 对于一个反映电子线路的图, 边上的权可表示两端点间的电阻、电流或电压; 对于一个反映零件装配的图, 边上的权可表示一个端点需要装配另一个端点的零件的数量; 对于一个反映工程进度的图, 边上的权可表示从前一子工程到后一子工程所需要的天数。边上带有权的图称作带权图, 也常称作网 (network)。如图 7-3 就是一个网。

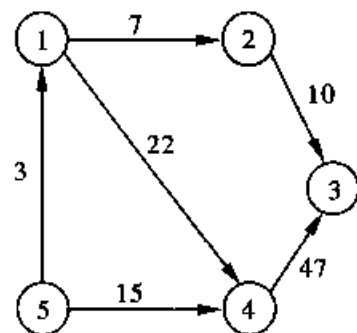


图 7-3 网的示例

### 7.3 图的存储结构

图的存储结构又称图的存储表示或图的表示。它有多种方法,这里主要介绍邻接矩阵、邻接表和边集数组这三种方法。

#### 一、邻接矩阵

邻接矩阵(adjacency matrix)是表示顶点之间相邻关系的矩阵。设  $G = (V, E)$  是具有  $n$  个顶点的图,顶点序号依次为  $1, 2, \dots, n$ , 则  $G$  的邻接矩阵是具有如下定义的  $n$  阶方阵。

$$A_{ij} = \begin{cases} 1, & \text{对于无向图: } (v_i, v_j) \text{ 或 } (v_j, v_i) \in E(G) \\ & \text{对于有向图: } \langle v_i, v_j \rangle \in E(G) \\ 0, & \text{反之} \end{cases}$$

若图  $G$  是一个带权图,则用邻接矩阵表示也很方便,只要把 1 换为相应边上的权值,把 0 换为  $\infty$  即可。其中  $\infty$  表示“无穷大”,实际存储时它要大于图  $G$  中需要进行的各种运算所得到的最大的权值。

采用邻接矩阵表示图,便于查找图中任一条边或边上的权。如要查找边  $(v_i, v_j)$  或  $\langle v_i, v_j \rangle$ , 则只要查找邻接矩阵中第  $i$  行第  $j$  列的元素  $A_{ij}$  是否非零(或非  $\infty$ )即可;若该元素非零(或非  $\infty$ ), 则表明此边存在,否则此边不存在。因此邻接矩阵中的元素可以随机存取,所以其查找时间复杂性为  $O(1)$ 。如要查找  $v_i$  的一个邻接点(对于无向图)或出边邻接点(对于有向图), 则只要在第  $i$  行上查找出一个非零(或非  $\infty$ )元素。以该元素所在的列号  $j$  为序号的顶点  $v_j$  就是所求的一个邻接点或出边邻接点。一般算法要求是依次查找一个顶点  $v_i$  的所有邻接点(对于有向图为出边邻接点或入边邻接点), 此时需访问对应第  $i$  行或第  $i$  列上的所有元素,所以其时间复杂性为  $O(n)$ 。

图的邻接矩阵存储需要占用  $n \times n$  个存储单元,所以其空间复杂性为  $O(n^2)$ 。这种存储结构用于表示稠密图能够充分利用存储空间,但若用于表示稀疏图,则致使邻接矩阵变为稀疏矩阵,从而造成存储空间的很大浪费。

例如图 7-3 的邻接矩阵表示如下:

	1	2	3	4	5
1	0	7	$\infty$	22	$\infty$
2	$\infty$	0	10	$\infty$	$\infty$
3	$\infty$	$\infty$	0	$\infty$	$\infty$
4	$\infty$	$\infty$	47	0	$\infty$
5	3	$\infty$	$\infty$	15	0

图的邻接矩阵表示很容易生成,这是一个生成无向带权图的算法描述:

Procedure Create (G)

```

for i ← 1 to n do
    for j ← 1 to n do
         $G_{i,j} \leftarrow \infty$ 
    for k ← 1 to e do
        Input (i, j, w)
         $G_{i,j} \leftarrow w$ 
         $G_{j,i} \leftarrow w$ 
    
```

## 二、邻接表

邻接表 (adjacency list) 是对图中的每个顶点建立一个邻接关系的单链表, 并把它们的表头指针用向量存储的一种图的表示方法。为顶点  $v_i$  建立的邻接关系的单链表又称作  $v_i$  的邻接表。 $v_i$  邻接表中的每个结点用来存储以该顶点为端点或起点的一条边的信息, 因而被称为边结点。 $v_i$  邻接表中的结点数, 对于无向图来说, 等于  $v_i$  的边数、邻接点数或度数; 对于有向图来说, 等于  $v_i$  的出边数、出边邻接点数或出度数。边结点的类型通常被定义为三个域: 一是邻接点域 (adjvex), 用以存储顶点  $v_i$  的一个邻接顶点  $v_j$  的序号  $j$ ; 二是权域 (weight), 用以存储边  $(v_i, v_j)$  或  $\langle v_i, v_j \rangle$  上的权; 三是链域 (next), 用以链接  $v_i$  邻接表中的下一个结点。在这三个域中, 邻接点域和链域是必不可少的, 权域可根据情况取舍, 若表示的是无权图, 则可省去此域。对于每个顶点  $v_i$  的邻接表, 需要设置一个表头结点, 该结点除了包括  $v_i$  邻接表的表头指针域 (link) 外, 通常还包括用于存储顶点  $v_i$  信息的值域 (data), 若顶点  $v_i$  的值就是该顶点的编号  $i$ , 则此域可以省去。如图  $G$  中有  $n$  个顶点, 则就有  $n$  个表头结点, 为了便于随机访问任一顶点的邻接表, 需把这  $n$  个表头结点用一个向量 (即一维数组) 存储起来, 其中第  $i$  个分量存储  $v_i$  邻接表的表头结点。这样, 图  $G$  就可以由这个表头向量来表示。

在图的邻接表中便于查找一个顶点的边 (出边) 或邻接点 (出边邻接点), 这只要首先从表头向量中取出对应的表头指针, 然后从表头指针出发进行查找即可。由于每个顶点单链表的平均长度为  $e/n$  (对于有向图) 或  $2e/n$  (对于无向图), 所以此查找运算的时间复杂性为  $O(e/n)$ 。但要从有向图的邻接表中查找一个顶点的入边或入边邻接点, 那就不方便了, 它需要扫描所有顶点邻接表中的边结点, 因此其时间复杂性为  $O(n+e)$ 。对于那些需要经常查找顶点入边或入边邻接点的运算, 可以为此专门建立一个逆邻接表 (contrary adjacency list), 该表中每个顶点的单链表不是存储该顶点的所有出边的信息, 而是存储所有入边的信息。

图的邻接表表示和图的邻接矩阵表示, 虽然方法不同, 但也存在着对应的关系。邻接表中每个顶点  $V_i$  的单链表对应邻接矩阵中的第  $i$  行, 整个邻接表可看作是邻接矩阵的带行指针向量的链接存储, 整个逆邻接表可看作是邻接矩阵的带列指针向量的链接存储。我们知道, 对于稀疏矩阵, 若采用链接存储是比较节省存储空间的, 所以稀疏图的邻接表表示比邻接矩阵表示要节省存储空间。

例如图 7-3 的邻接表表示如图 7-4 所示。

下面给出一个生成无向带权图的邻接表的算法描述:

Const

$n \leftarrow$  图顶点数

$e \leftarrow$  图中边数

Type

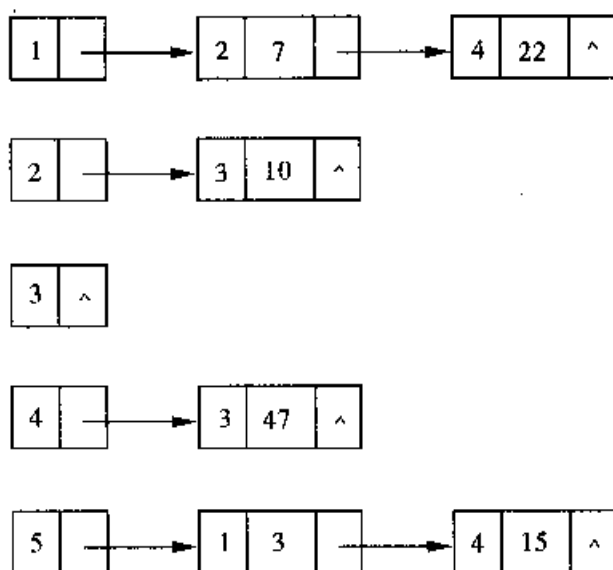


图 7-4 链接存储

```

Edgeptr = ^ edgenode;
Edgenode = record
    Adjvex : 1..n | 邻接点域 |
    Weight : WeightType | 权值 |
    Next : Edgeptr | 链域 |
End
Vexnode = record
    Data : DataType
    Link : edgeptr
End
Adjlish = array [1..n] of vexnode

```

```

Procedure Create (G)
  For l ← 1 to n do
    Input (Gi.data)
    Gi.link ← nil
  For k ← 1 to e do
    Input (l, j)
    GetMemory (s)
    s^.adjvex ← j
    s^.next ← Gi.link
    Gi.link ← s
  
```

### 三、边集数组

边集数组 (edgeset array) 是利用一维数组存储图中所有边的一种图的表示方法。该数组中所

含元素的个数要大于等于图中边的条数，每个元素用来存储一条边的起点、终点（对于无向图，可选定边的任一端点为起点或终点）和权（若有的话），各边在数组中的次序可任意安排，也可根据具体要求而定。

例如图 7-3 的边集数组表示如下：

起点	终点	权值
1	2	10
1	4	22
2	3	10
4	3	47
5	1	3
5	1	15

下面给出一个生成无向带权图的边集数组的算法描述：

Procedure Creat (G)

For k  $\leftarrow$  1 to n do

Input (I, J, w)

$G_k$ .fromvex  $\leftarrow$  I

$G_k$ .endvex  $\leftarrow$  J

$G_k$ .weight  $\leftarrow$  w

在边集数组中查找一条边或一个顶点的度都需要扫描整个数组，所以其时间复杂性为  $O(e)$ 。边集数组适合那些对边依次进行处理的运算，不适合对顶点的运算和对任一条边的运算。空间复杂性为  $O(e)$ ，从空间复杂性上讲，边集数组也适于表示稀疏图。

图的邻接矩阵、邻接表和边集数组表示各有利弊，具体应用时，要根据图的稠密和稀疏程度以及算法的要求进行选择。

## 7.4 图的遍历

图的遍历就是从指定的某个顶点（称为初始点）出发，按照一定的搜索方法对图中的所有顶点各作一次访问的过程。图的遍历比树的遍历要复杂，因为从树根到达树中的每个结点只有一条路径，而从图的初始点到达图中的每个顶点可能存在着多条路径。当顺着图中的一条路径访问过某一顶点后，可能还会顺着另一条路径回到该顶点。为了避免重复访问图中的同一个顶点，必须用标志量来标记每个顶点是否被访问过。





根据搜索方法的不同,图的遍历有两种:一种叫做深度优先搜索遍历,另一种叫做广度优先搜索遍历。

### 一、深度优先遍历

深度优先搜索 (depth first search) 遍历类似树的先根遍历,它是一个递归过程,可叙述为:首先访问一个顶点  $v_i$  (开始为初始点),并将其标记为已访问过,然后从  $v_i$  的一个未被访问的邻接点 (无向图) 或出边邻接点 (有向图) 出发进行深度优先搜索遍历,当  $v_i$  的所有邻接点均被访问过时,则退回到上一个顶点  $v_k$ ,从  $v_k$  的另一个未被访问过的邻接点出发进行深度优先搜索遍历。

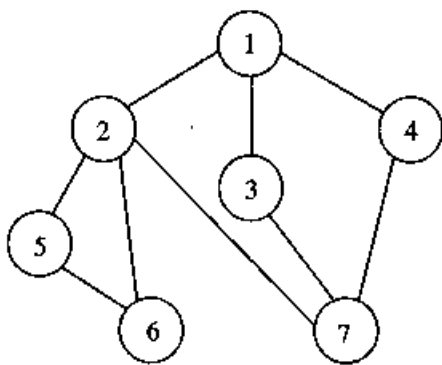


图 7-5 图的示例

如图 7-5 的深度优先遍历的具体步骤如下:

(1) 访问顶点  $v_1$ , 并标记  $v_1$  已被访问过。选取  $v_1$  的任意一个未访问过的邻接点 (假设选  $v_2$ ) 进行深度优先遍历。

(2) 访问顶点  $v_2$ , 并标记  $v_2$  已被访问过。选取  $v_2$  的任意一个未访问过的邻接点 (假设选  $v_5$ ) 进行深度优先遍历。

(3) 访问顶点  $v_5$ , 并标记  $v_5$  已被访问过。选取  $v_5$  的任意一个未访问过的邻接点 (假设选  $v_6$ ) 进行深度优先遍历。

(4) 访问顶点  $v_6$ , 并标记  $v_6$  已被访问过。 $v_6$  邻接点都被访问过,回溯到  $v_2$ , 选取  $v_7$  进行深度优先遍历。

(5) 访问顶点  $v_7$ , 并标记  $v_7$  已被访问过。选取  $v_7$  的任意一个未访问过的邻接点 (假设选  $v_3$ ) 进行深度优先遍历。

(6) 访问顶点  $v_3$ , 并标记  $v_3$  已被访问过。 $v_3$  邻接点都被访问过,回溯到  $v_1$ , 选取  $v_4$  进行深度优先遍历。

(7) 访问顶点  $v_4$ , 并标记  $v_4$  已被访问过。 $v_4$  邻接点都被访问过,回溯到  $v_4, v_2, v_1$  发现所有点的邻接点都被访问过,至此遍历结束。

得到的深度优先遍历序列为:  $v_1, v_2, v_5, v_6, v_7, v_3, v_4$ 。

下面给出用邻接矩阵存储的图的深度优先搜索的算法描述:

Procedure Dfs ( I )

Print ( I )

visited<sub>i</sub> ← true

for j ← 1 to n do

if not visitedj and (gi, j = 1) then dfs (j)

图的深度优先遍历因为选取的点的顺序和边的顺序的不同,不是唯一的。对邻接矩阵表示的图进行深度优先搜索遍历时,需要扫描邻接矩阵中的每一个元素,所以其时间复杂性为  $O(n^2)$ 。对邻接表表示的图进行深度优先搜索遍历时,需要扫描邻接表中的每个边结点,所以其时间复杂性为  $O(e)$ 。

## 二、广度优先搜索遍历

广度优先搜索 (breadth-first search) 遍历类似树的按层遍历,其过程为:首先访问初始点  $v_1$ ,并将其标记为已访问过,接着访问  $v_1$  的所有未被访问过的邻接点  $v_{11}, v_{12}, \dots, v_{1n}$  并均标记为已访问过,然后再按照  $v_{11}, v_{12}, \dots, v_{1n}$  的次序,访问每一个顶点的所有未被访问过的邻接点,并均标记为已访问过,依此类推,直到图中所有和初始点  $v_1$  有路径相通的顶点都被访问过为止。

在广度优先搜索遍历中,先被访问的顶点,其邻接点亦先被访问,所以在算法的实现中需要使用一个队列,用来依次记住被访问过的顶点。算法开始时,将初始点  $v_1$  访问后插入队列中,以后每从队列中删除一个元素,就依次访问它的每一个未被访问过的邻接点,并令其进队,这样,当队列为空时表明所有与初始点有路径相通的顶点都已访问完毕,算法到此结束。

如图 7-5 的广度优先遍历的具体步骤如下:

- (1) 访问顶点  $v_1$ , 并标记已被访问过。
- (2) 访问顶点  $v_1$  的所有未被访问过的邻接点  $v_2, v_3, v_4$ , 并标记已访问过。
- (3) 访问顶点  $v_2$  的所有未被访问过的邻接点  $v_5, v_6, v_7$ , 并标记已访问过。
- (4) 依次访问  $v_3, v_4, v_5, v_6, v_7$  所有未被访问过的邻接点,发现已经都被访问,广度优先遍历结束。

得到的广度优先遍历序列为  $v_1, v_2, v_3, v_4, v_5, v_6, v_7$ 。

下面给出用邻接矩阵存储的图的广度优先搜索的算法描述:

Procedure Bfs ( I )

Setnull ( Q ) | 置对空 |

Print ( I )

Visitedi  $\leftarrow$  true

Insert ( Q, i ) | I 入队 |

Repeat

K  $\leftarrow$  delete ( Q ) | 队首元素出队 |

For j  $\leftarrow$  1 to n do

If ( gk, j = 1 ) and not visitedj then

Print ( j )

Visitedj  $\leftarrow$  true

Insert ( Q, j )

Until empty ( Q ) | 队为空 |

与图的深度优先搜索遍历一样,对于图的广度优先搜索遍历,若采用邻接矩阵表示,其时间

复杂性为  $O(n^2)$ ；若采用邻接表表示，其时间复杂性为  $O(e)$ 。由图的某个顶点出发进行广度优先搜索遍历时，访问各顶点的次序不同也会导致搜索序列的不同，所以也不是唯一的。

深度优先遍历和广度优先遍历得到的序列各有特点，比如深度优先序列构造的生成树没有横向弧，广度优先序列构造的生成树可以很容易得到每个结点的层次等，需要具体情况具体分析，选取一种合适的。

### 三、非连通图的遍历

前面提到的深度优先遍历和广度优先遍历都只从图的一个顶点开始进行一次遍历，对于连通图可以遍历到图的所有结点，但如果图不连通，则有一部分结点无法访问到。修改很简单，每次选取任意一个没有被遍历过的结点开始一次遍历，重复此操作直到遍历完图的所有结点即可。

## 7.5 图的生成树与最小生成树

在一个连通图  $G$  中，如果取它的全部顶点和一部分边构成一个子图  $G'$ ，即：

$$V(G') = V(G) \text{ 和 } E(G') \subseteq E(G)$$

若边集  $E(G')$  中的边既将图中的所有顶点连通又不形成回路，则称子图  $G'$  是原图  $G$  的一棵生成树。

下面简单说明一下既包含连通图  $G$  中的全部  $n$  个顶点又没有回路的子图  $G'$ （即生成树）必含有  $n-1$  条边。要构造子图  $G'$ ，首先从图  $G$  中任取一个顶点加入  $G'$  中，此时  $G'$  中只有一个顶点，自然是连通的，以后每次添加一条一个端点在图  $G'$  中，另一个不在  $G'$  中的边，并将不在  $G'$  中的端点连通到图  $G'$  中，这样不会产生回路。 $n-1$  次后，就向  $G'$  中加入了  $n-1$  条边和  $n-1$  个顶点，使得  $G'$  中  $n$  个点连通且不存在回路。

图 7-6 中的 (b), (c), (d) 均是图 (a) 的生成树。对于一个连通网（假设边上的权都非负）生成树的不同，树的权（即树中所有边上的权值总和）。其中权最小的生成树为图的最小生成树（minimum spanning tree）。

求图的最小生成树很有实际意义，例如，若一个连通网表示城市之间的通讯系统，网的顶点代表城市，网的边代表城市之间架设通讯线路的造价，各城市之间的距离不同，地理条件不同，其造价也不同，即边上的权不同，现在要求既要连通所有城市、又要使总造价最低，这就是一个求图的最小生成树的问题。

下面讨论求图的最小生成树的两种算法：普里姆（Prim）算法和克鲁斯卡尔（Kruskal）算法。

### 一、普里姆算法

假设  $G = (V, E)$  是一个具有  $n$  个顶点的连通网， $T = (U, TE)$  是  $G$  的最小生成树，其中  $U$  是  $T$  的顶点集， $TE$  是  $T$  的边集， $U$  和  $TE$  的初值均为空集。算法开始时，首先从  $V$  中任取一个顶点（假定取  $v_1$ ），将它并入  $U$  中，此时  $U = \{v_1\}$ ，然后只要  $U$  是  $V$  的真子集（即  $U \subset V$ ），就从那些其一个端点已在  $T$  中，另一个端点仍在  $T$  外的所有边中，找一条最短（即权值最小）边，假定为  $(v_i, v_j)$ ，其中  $v_i \in U, v_j \in V - U$ ，并把该边  $(v_i, v_j)$  和顶点  $v_j$  分别并入  $T$  的边集  $TE$  和顶

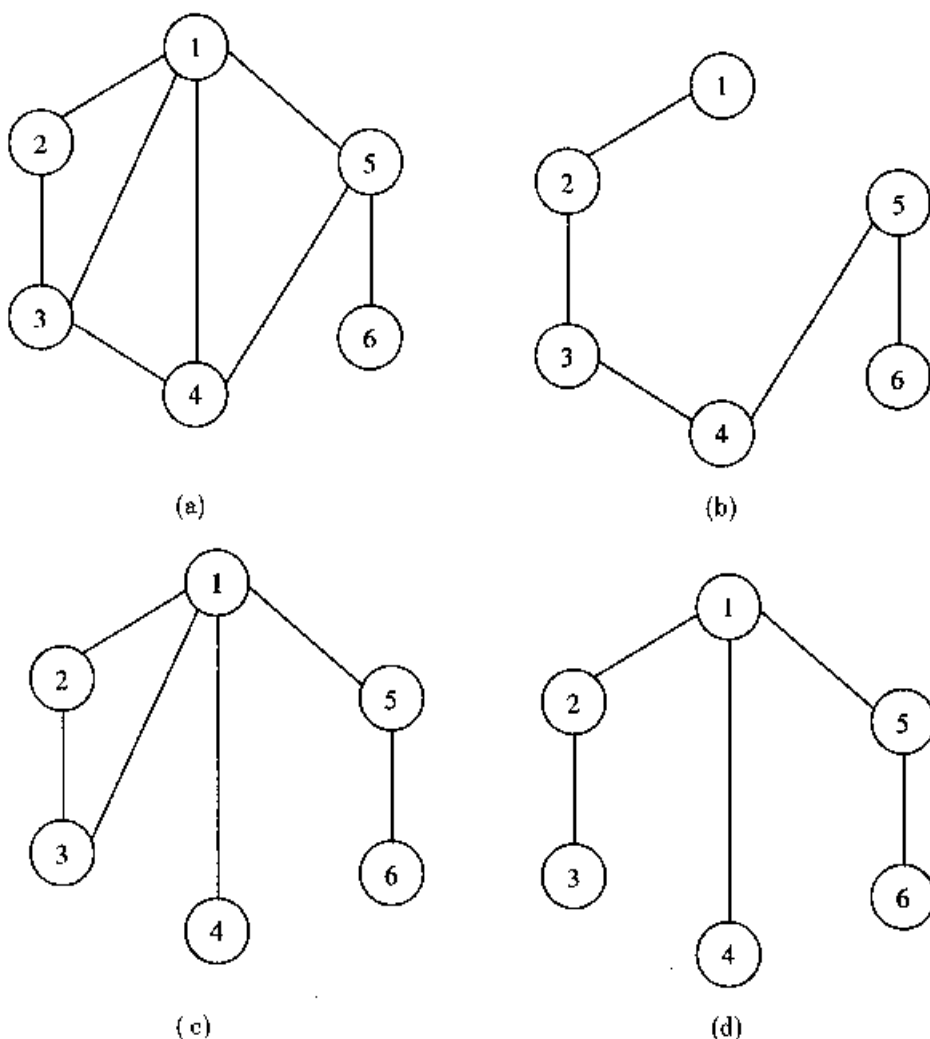


图 7-6 生成树的示例

点集  $U$ ，如此进行下去，每次往生成树里并入一个顶点和一条边，直到  $(n-1)$  次后就把所有  $n$  个顶点都并入到生成树  $T$  的顶点集中，此时  $U=V$ ， $TE$  中含有  $(n-1)$  条边， $T$  就是最后得到的最小生成树。

普里姆算法的关键之处是：每次如何从生成树  $T$  中到  $T$  外的所有边中，找出一条最短边。例如，在第  $k$  次前，生成树  $T$  中已有  $k$  个顶点和  $(k-1)$  条边，此时  $T$  中到  $T$  外的所有边数为  $k(n-k)$ ，当然它包括两顶点间没有直接边相连，其权值被看作为“无穷大”的边在内，从如此多的边中查找最短边，其时间复杂性为  $O(k(n-k))$ ，显然是很费时的。是否有一种好的方法能够降低查找最短边的时间复杂性呢？回答是肯定的，它能够使查找最短边的时间复杂性降低到  $O(n-k)$ 。方法是：假定在进行第  $k$  次前已经保留着从  $T$  中到  $T$  外每一顶点（共  $(n-k)$  个顶点）的各一条最短边，进行第  $k$  次时，首先从这  $(n-k)$  条最短边中，找出一条最最短的边（它就是从  $T$  中到  $T$  外的所有边中的最短边），假设为  $(v_i, v_j)$ ，此步需进行  $(n-k)$  次比较；然后把边  $(v_i, v_j)$  和顶点  $v_j$  分别并入  $T$  中的边集  $TE$  和顶点集  $U$  中，此时  $T$  外只有  $n-(k+1)$  个顶点，对于其中的每个顶点  $v_i$ ，若  $(v_j, v_i)$  边上的权值小于已保留的从原  $T$  中到  $v_i$  的最短边的权值，则用  $(v_j, v_i)$  修改之，使从  $T$  中到  $T$  外顶点  $v_i$  的最短边为  $(v_j, v_i)$ ，否则原有最短边保持不变。这样，就

把第  $k$  次后从  $T$  中到  $T$  外每一顶点  $v_i$  的各一条最短边都保留下来了。为进行第  $(k+1)$  次运算做好了准备, 此步需进行  $(n-k-1)$  次比较。所以, 利用此方法求第  $k$  次的最短边共需比较  $2(n-k)-1$  次, 即时间复杂性为  $O(n-k)$ 。

所以普里姆算法总的时间复杂性为  $O(n^2)$ 。

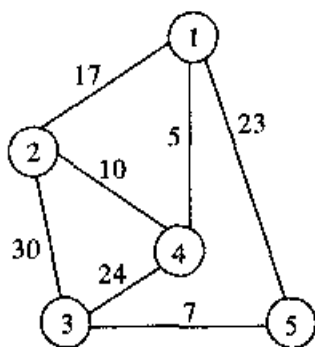


图 7-7 图的示例

下面给出图 7-7 用普里姆算法求最小生成树的具体步骤, 其中  $T$  表示生成树中顶点集,  $TE$  表示生成树中边集,  $\min$  表示到  $T$  中的最短距离的边。

$$(1) T = \{v_1\}$$

$$TE = \{\}$$

$$\min = \{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_1, v_5)\}$$

如图 7-8 (a)

$$(2) T = \{v_1, v_4\}$$

$$TE = \{(v_1, v_4)\}$$

$$\min = \{(v_4, v_2), (v_4, v_3), (v_1, v_5)\}$$

如图 7-8 (b)

$$(3) T = \{v_1, v_4, v_2\}$$

$$TE = \{(v_1, v_4), (v_4, v_2)\}$$

$$\min = \{(v_4, v_3), (v_1, v_5)\}$$

如图 7-8 (c)

$$(4) T = \{v_1, v_4, v_2, v_5\}$$

$$TE = \{(v_1, v_4), (v_4, v_2), (v_1, v_5)\}$$

$$\min = \{(v_5, v_3)\}$$

如图 7-8 (d)

$$(5) T = \{v_1, v_4, v_2, v_5, v_3\}$$

$$TE = \{(v_1, v_4), (v_4, v_2), (v_1, v_5), (v_5, v_3)\}$$

$$\min = \{\}$$

如图 7-8 (e)。

下面给出普里姆算法的算法描述:

Procedure Prim (G)

for  $i \leftarrow 2$  to  $n$  do

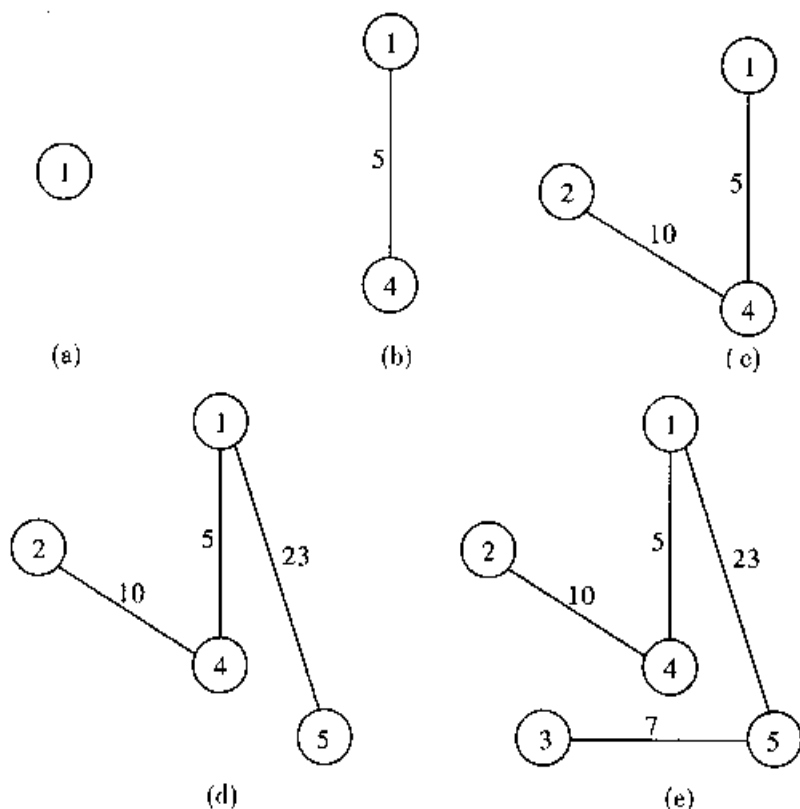


图 7-8 普里姆算法示例

```

 $\min_i \leftarrow g_{1,i}$ , | 将每个 T 外的点到 T 中的最短距离初始化;
 $Fa_i \leftarrow 1$ 
for  $k \leftarrow 2$  to  $n$  do
     $i \leftarrow \text{FindShortest}$  | 找 T 外的点到 T 中的最短路径最短的点 |
    if  $i = 0$  then return (no solution);
     $T \leftarrow T \cup i$ 
     $TE \leftarrow TE \cup (i, Fa_i)$ 
    for  $j \leftarrow 2$  to  $n$  do
        if not ( $j \in T$ ) and ( $g_{i,j} < \min_j$ ) then
             $\min_j \leftarrow g_{i,j}$ 
             $Fa_j \leftarrow i$ 
Return (TE)
    
```

## 二、克鲁斯卡尔算法

假设  $G = (V, E)$  是一个具有  $n$  个顶点的连通网,  $T = (U, TE)$  是  $G$  的最小生成树,  $U$  的初值等于  $V$ , 即包含有  $G$  中的全部顶点,  $TE$  的初值为空。此算法的基本思想是, 将图  $G$  中的边按权值从小到大的顺序依次选取, 若选取的边使生成树  $T$  不形成回路, 则把它并入  $TE$  中, 保留作为  $T$  的一条边; 若选取的边使生成树  $T$  形成回路, 则将其舍弃。如此进行下去, 直到  $TE$  中包含有  $n-1$  条边为止。此时的  $T$  即为最小生成树。

克鲁斯卡尔算法的关键之处是：如何判断欲加入的一条边是否与生成树中已选取的边形成回路。这可将各顶点划分为所属集合的方法来解决，每个集合中的顶点表示一个无回路的连通分量。算法开始时，由于生成树的顶点集等于图  $G$  的顶点集，边集为空，所以  $n$  个顶点分属于  $n$  个集合。每个集合中只有一个顶点，表明顶点之间互不连通。

当从边集数组中按次序选取一条边时，若它的两个端点分属于不同的集合，则表明此边连通了两个不同的连通分量，因每个连通分量无回路，所以连通后得到的连通分量仍不会产生回路，此边应选取作为生成树的一条边，同时把端点所在的两个集合合并成一个，即成为一个连通分量，当选取的一条边的两个端点同属于一个集合时，此边应放弃，因同一个集合中的顶点是连通无回路的，若再加入一条边则必产生回路。

克鲁斯卡尔算法在对边排序时如果用快速排序的话时间复杂性为  $O(E \log_2 E)$ ，而添边的过程可以用并查集实现，复杂性为  $O(E \alpha(E))$ ，所以总的时间复杂性为  $O(E \log_2 E + E \alpha(E))$ 。

下面给出图 7-7 用克鲁斯卡尔算法求最小生成树的具体步骤，其中  $U$  表示顶点的集合， $TE$  表示生成树中的边。

$$(1) \quad U = \{v_1, v_2, v_3, v_4, v_5\}$$

$$TE = \{\}$$

如图 7-9 (a)

$$(2) \quad U = \{v_1 v_4, v_2, v_3, v_5\}$$

$$TE = \{(v_1, v_4)\}$$

如图 7-9 (b)

$$(3) \quad U = \{v_1 v_4, v_2, v_3 v_5\}$$

$$TE = \{(v_1, v_4), (v_3, v_5)\}$$

如图 7-9 (c)

$$(4) \quad U = \{v_1 v_4 v_2, v_3 v_5\}$$

$$TE = \{(v_1, v_4), (v_3, v_5), (v_4, v_2)\}$$

如图 7-9 (d)

$$(5) \quad U = \{v_1 v_4 v_2 v_3 v_5\}$$

$$TE = \{(v_1, v_4), (v_3, v_5), (v_4, v_2), (v_1, v_5)\}$$

如图 7-9 (e)

下面给出克鲁斯卡尔算法的算法描述：

Procedure Kruskal ( $G$ )

SortEdge

└将边按权值升序排列┘

For  $I \leftarrow 1$  to  $e$  do

    If Check (Edge $_I$ )

        └判断第  $I$  条边的两个端点是否在一个连通图中┘

    then

$TE \leftarrow TE \cup \text{Edge}_I$

        Merge (Edge $_I$ ) └合并第  $I$  条边两个端点所在集合┘

$m \leftarrow m + 1$

        if  $m = n$  then return (TE)

return (no solution)

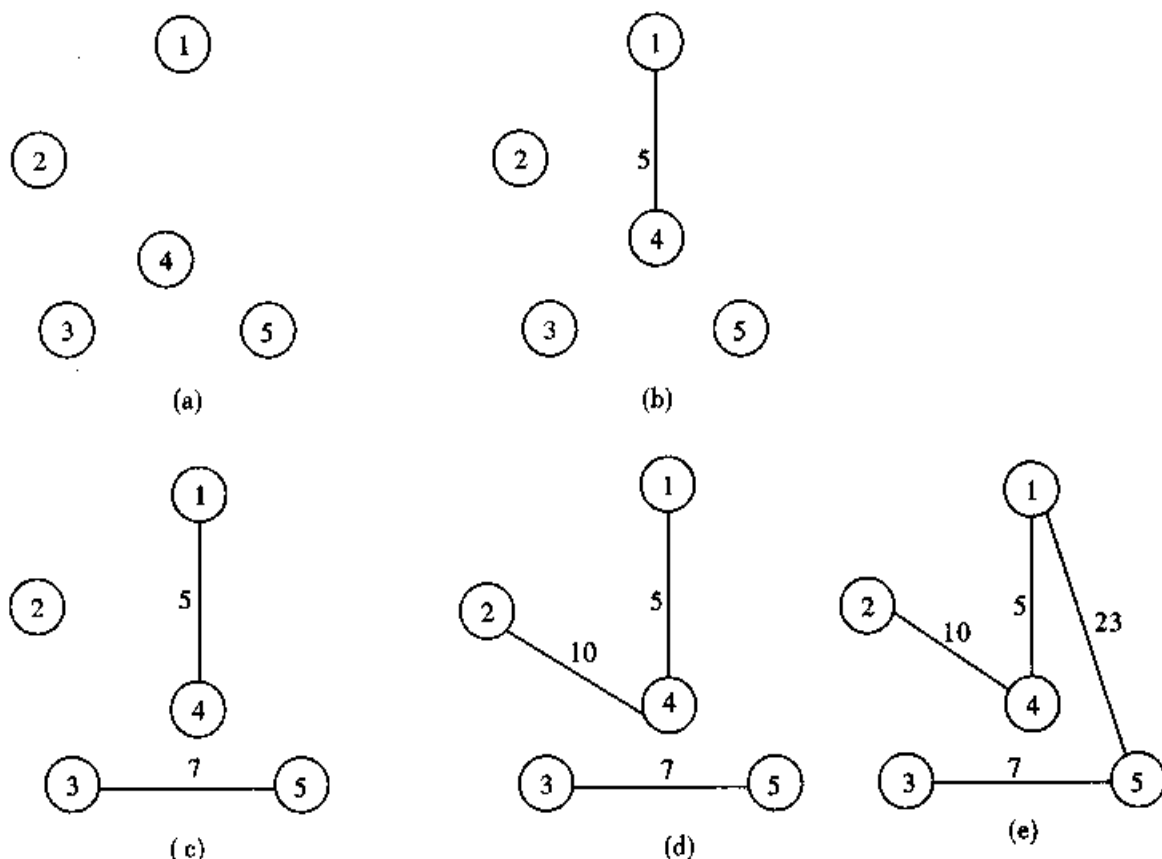


图 7-9 克鲁斯卡尔算法示例

### 例题 7-1 雅典奥运修筑天桥。

#### 问题描述：

2004 年奥运会在希腊雅典举行，现在已经在雅典城建了许多新的大型体育馆。为了使观众能够方便地在体育馆之间通行，雅典政府决定在体育馆之间建一些天桥，以使得其中任意两个体育馆之间都有直接或间接的天桥相连。当然，从经济的角度出发，政府希望所有的修天桥费用之和最小，其中修天桥费用与天桥的长度成正比，所以希望建设的天桥的总长度尽量小。

#### 输入：

第一行是一个整数  $N$  ( $N \leq 100$ )，表示雅典城内体育馆的数目。第二行至第  $N+1$  行每行两个数  $X_i, Y_i$  ( $0 \leq X_i, Y_i \leq 100$ )，表示第  $i$  个体育馆的坐标。

#### 输出：

只有一个数，表示天桥总长度最小值。

#### 分析：

将  $n$  个体育馆每两个之间连一条边，构造一个有  $n$  个顶点的完全图，每两个顶点之间边上的权值为它们之间的平面距离，显然，解题的关键是求这个图中的最小生成树，求出后，最短长度即可得。这是应用最小生成树解题的典型。可以用普里姆或者克鲁斯卡尔来解决，但由于是完全图，所以普里姆算法的效率更高。

#### 源程序：





```
const
  fin = 'input.txt';
  fon = 'output.txt';
  maxn = 100;

var
  g : array [1..maxn, 1..maxn] of real;
  x, y, len : array [1..maxn] of real;
  select : array [1..maxn] of boolean;
  n : integer;
  ans : real;

procedure init;
var
  i : integer;
begin
  assign (input, fin); reset (input);
  readln (n);
  for i : = 1 to n do read (x [i], y [i]);
  close (input);
end;

function dist (i, j : integer) : real;
begin
  dist : = sqrt (sqr (x [i] - x [j]) + sqr (y [i] - y [j]));
end;

procedure prepare;
var
  i, j : integer;
begin
  for i : = 1 to n do
    for j : = 1 to n do
      g [i, j] : = dist (i, j);
    end;
  end;

procedure main;
var
  i, j, k : integer;
  min : real;
```



```

begin
  for i : = 1 to n do len [i] : = g [1, i];
  for k : = 1 to n do begin
    min : = 1e20;
    for j : = 1 to n do
      if not select [j] and (len [j] < min) then begin
        min : = len [j];
        i : = j;
      end;
    ans : = ans + min;
    select [i] : = true;
    for j : = 1 to n do
      if not select [j] and (g [i, j] < len [j]) then
        len [j] : = g [i, j];
    end;
  end;

procedure print;
begin
  assign (output, fon); rewrite (output);
  writeln (ans : 0 : 2);
  close (output);
end;

begin
  init;
  prepare;
  main;
  print;
end.

```

## 7.6 最短路径

由图的概念可知, 在一个无权图中, 若从一顶点到另一顶点存在着一条路径 (这里只讨论无回路的简单路径), 则称该路径长度为该路径上所经过的边的数目, 它也等于该路径上的顶点数减 1。由于从一顶点到另一顶点可能存在着多条路径, 每条路径上所经过的边数可能不同, 即路径长度不同, 我们把路径长度最短 (即经过的边数最少) 的那条路径叫做最短路径, 其路径长度叫做最短路径长度或最短距离。求图中一顶点  $v_i$  到其余各顶点的最短路径和最短距离比较容易, 只要

从该顶点  $v_i$  出发对图进行一次广度优先搜索遍历, 在遍历时记下每个结点的层次即可。

若图是带权图(假定权值非负)从源点  $v_i$  到终点  $v_j$  的每条路径上的权(它等于该路径上所经边上的权值之和, 称为该路径的带权路径长度)可能不同, 我们把权值最小的那条路径也称做最短路径, 其权值也称作最短路径长度或最短距离。

实际上, 这两类最短路径问题可合并为一类, 这只要把第一类的每条边的权都设为 1 就归属于第二类了, 所以在以后的讨论中, 若不特别指明, 均是指第二类的最短路径问题。求图的最短路径问题用途很广。例如, 若用一个图表示城市之间的运输网, 图的顶点代表城市, 图上的边表示两端点对应城市之间存在着运输线, 边上的权表示该运输线上的运输时间或单位重量的运费, 考虑到两城市间海拔高度不同、流水方向不同等因素, 将造成来回运输时间或运费的不同, 所以这种图通常是一个有向图。如何能够使从一城市到另一城市的运输时间最短或者运费最省呢? 这就是一个求两城市间的最短路径问题。

求图的最短路径问题包括两个子问题: 一是求图中一顶点到其余各顶点的最短路径, 二是求图中每对顶点之间的最短路径。下面分别进行讨论。

### 一、从一顶点到其余各顶点的最短路径

对于一个具有  $n$  个顶点和  $e$  条边的图  $G$ , 从某一顶点(即源点)  $v_i$  到其余任一顶点(即终点)  $v_j$  的最短路径, 可能是它们之间的边  $(v_i, v_j)$  或  $\langle v_i, v_j \rangle$ , 也可能是经过  $k$  个  $(1 \leq k \leq n-2)$ , 最多经过除源点和终点之外的所有顶点(中间顶点和  $k+1$  条边)所形成的路径。

那么, 如何求出从源点  $v_i$  到其余每一个顶点的最短路径呢? 迪杰斯特拉(Dijkstra)于 1959 年提出了解决此问题的一般算法, 具体做法是按照从源点到其余每一顶点的最短路径长度的升序依次求出从源点到各顶点的最短路径及长度, 每次求出从源点  $v_i$  到一个终点  $v_j$  的最短路径及长度后, 都要以  $v_j$  作为新考虑的中间点, 用  $v_i$  到  $v_j$  的最短路径和最短路径长度对  $v_i$  到其他尚未求出最短路径的那些终点的当前路径及长度作必要的修改, 使之成为当前新的最短路径和最短路径长度, 当进行  $n-2$  次后算法结束。

迪杰斯特拉算法需要设置一个集合(假定为  $S$ ), 其作用是保存已求得最短路径的终点, 它的初值中只有一个元素, 即源点  $v_i$ , 以后每求出一个从源点  $v_i$  到终点  $v_j$  的最短路径, 就将该顶点  $v_j$  并入  $S$  集合中, 以便作为新考虑的中间点; 还需要设置一个数组  $\text{dist}(1..n)$ , 该数组中的第  $j$  个元素  $\text{dist}_j$  用来保存从源点  $v_i$  到终点  $v_j$  的目前最短路径长度, 它的初值为  $(v_i, v_j)$  或  $\langle v_i, v_j \rangle$  边上的权值; 若  $v_i$  到  $v_j$  没有边, 则权值为无穷大, 以后每考虑一个新的中间点时,  $\text{dist}_j$  的值可能变小; 另外, 再设置一个与  $\text{dist}$  数组相对应的数组  $\text{path}(1..n)$ , 该数组中的第  $j$  个元素  $\text{path}_j$  用来保存与  $\text{dist}_j$  相对应的目前最短路径, 它的初值为  $v_i$  到  $v_j$  的边, 若不存在边则为空。

此算法的执行过程是: 首先从  $S$  集合以外的顶点(即待求出最短路径的终点)所对应的  $\text{dist}$  数组元素中, 查找出其值最小的元素(假定为  $\text{dist}_m$ ), 该元素值就是从源点  $v_i$  到终点  $v_m$  的最短路径长度(证明从略), 对应  $\text{path}$  数组中的元素  $\text{path}_m$  就是从  $v_i$  到  $v_m$  的最短路径(即经过的顶点序列或边的序列), 接着把已求得最短路径的终点  $v_m$  并入集合  $S$  中, 然后, 以  $v_m$  作为新考虑的中间点, 对  $S$  集合以外的每个顶点  $v_j$ , 比较  $\text{dist}_m + G_{m,j}$  与  $\text{dist}_j$  的大小, 若前者小则替换  $\text{dist}_j$ , 使  $\text{dist}_j$  始终保持到目前为止最短的路径长度, 同时用  $\text{path}_m$  并上  $v_j$  后替换  $\text{path}_j$ , 使之与  $\text{dist}_j$  的修改相对应; 重复  $n-2$  次上述运算过程, 即可在  $\text{dist}$  数组中得到从源点  $v_i$  到其余每个顶点的最短路径长度, 在  $\text{path}$  数组中得到相应的最短路径。

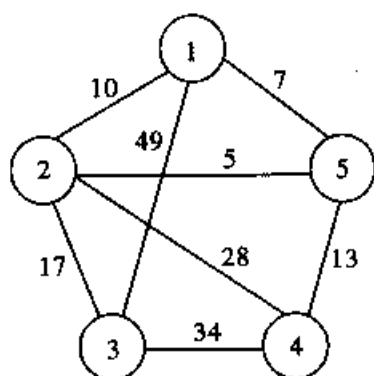


图 7-10 带权图示例

下面给出图 7-10 以  $v_1$  为起点的迪杰斯特拉算法具体步骤:

$S_i$  为 1 表示顶点  $v_i$  属于  $S$  集合, 等于 0 则表示  $v_i$  不属于  $S$  集合。

(1)

	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$
S	1	0	0	0	0
Dist	0	10	49	$\infty$	7
Path	$v_1$	$v_1, v_2$	$v_1, v_3$		$v_1, v_5$

(2)

	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$
S	1	0	0	0	1
Dist	0	10	49	20	7
Path	$v_1$	$v_1, v_2$	$v_1, v_3$	$v_1, v_5, v_4$	$v_1, v_5$

(3)

	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$
S	1	1	0	0	1
Dist	0	10	27	20	7
Path	$v_1$	$v_1, v_2$	$v_1, v_2, v_3$	$v_1, v_5, v_4$	$v_1, v_5$

(4)

	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$
S	1	1	0	1	1
Dist	0	10	27	20	7
Path	$v_1$	$v_1, v_2$	$v_1, v_2, v_3$	$v_1, v_5, v_4$	$v_1, v_5$

(5)

	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$
S	1	1	1	1	1
Dist	0	10	27	20	7
Path	$v_1$	$v_1, v_2$	$v_1, v_2, v_3$	$v_1, v_5, v_4$	$v_1, v_5$

下面给出无向图的迪杰斯特拉算法描述:

Procedure Dijkstra ( $G, \text{Start}$ )

For  $j \leftarrow 1$  to  $n$  do

If  $j \neq \text{start}$  then  $s_j \leftarrow 0$  else  $s_j \leftarrow 1$

$\text{Dist}_j \leftarrow g_{\text{start}, j}$

$\text{Path}_j \leftarrow \{\text{start} \mid V \mid j\}$

For  $k \leftarrow 1$  to  $n - 2$  do

$I \leftarrow \text{FindMin} \{ \text{找 } S \text{ 集合外 } \text{Dist} \text{ 最小的顶点} \}$

$S_i \leftarrow 1$

For  $j \leftarrow 1$  to  $n$  do

If  $(s_j = 0) \text{ and } (\text{dist}_i + g_{i,j} < \text{dist}_j)$  then

$\text{Dist}_j \leftarrow \text{dist}_i + g_{i,j}$

$\text{Path}_j \leftarrow \text{path}_i \vee \{j\}$

Return ( $\text{Dist}, \text{Path}$ )

## 二、每对顶点之间的最短路径

求图中每对顶点之间的最短路径是指把图中任意两个顶点  $v_i$  和  $v_j$  ( $i \neq j$ ) 之间的最短路径都计算出来。解决此问题有两种方法: 一是分别以图中的每个顶点为源点共调用  $n$  次迪杰斯特拉算法, 此方法的时间复杂性为  $O(n^3)$ ; 二是采用下面介绍的弗洛伊德 (Floyd) 算法, 此算法的时间复杂性仍为  $O(n^3)$ , 但比较简单。

弗洛伊德算法实际上是一个动态规划的算法。从图的邻接矩阵开始, 按照顶点  $v_1, v_2, \dots, v_n$  的次序, 分别以每个顶点  $v_k$  ( $1 \leq k \leq n$ ) 作为新考虑的中间点, 在第  $k-1$  次运算  $A^{k-1}$  ( $A^{(0)}$  为原图的邻接矩阵  $G$ ) 的基础上, 求出每对顶点  $v_i$  到  $v_j$  的最短路径长度  $A_{ij}^k$ 。计算公式为:

$$A_{i,j}^k = \begin{cases} G_{i,j} & k=0 \\ \min(A_{i,j}^{k-1}, A_{i,k}^{k-1} + A_{k,j}^{k-1}) & 1 \leq k \leq n \end{cases}$$

其中  $\min$  函数表示取参数表中的较小值。参数表中的前项表示在第  $k-1$  次运算后得到的  $v_i$  到  $v_j$  的目前最短路径长度，后项表示考虑以  $v_k$  作为新的中间点所得到的  $v_i$  到  $v_j$  的路径长度。若后项小于前项，则表明以包含  $v_k$  作为中间点的路径长度更短，所以更新  $A_{i,j}^k$ ，使  $A_{i,j}^k$  一直保存前  $k$  次运算后得到的从  $v_i$  到  $v_j$  的目前最短路径长度。当  $k$  从 1 取到  $n$  后，矩阵  $A^n$  就是最后得到的结果，其中  $A_{i,j}^n$  就是顶点  $v_i$  到  $v_j$  的最短路径长度。

下面给出图 7-11 用弗洛伊德算法求所有点对之间最短路径的具体步骤：

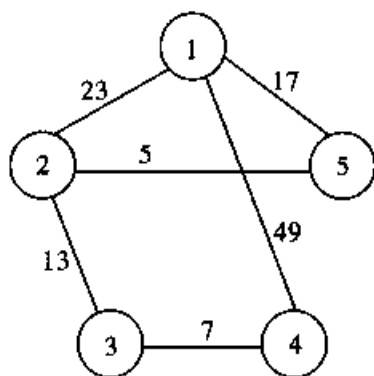


图 7-11 带权图示例

(1)  $k=0$

0	23	17	$\infty$	49
23	0	5	13	$\infty$
17	5	0	$\infty$	$\infty$
$\infty$	13	$\infty$	0	7
49	$\infty$	$\infty$	7	0

(2)  $k=1$

0	23	17	$\infty$	49
23	0	5	13	72
17	5	0	$\infty$	66
$\infty$	13	$\infty$	0	7
49	72	66	7	0

(3)  $k = 2$ 

0	23	17	36	49
23	0	5	13	72
17	5	0	18	66
36	13	18	0	7
49	72	66	7	0

(4)  $k = 3$ 

0	22	17	35	49
22	0	5	13	71
17	5	0	18	66
35	13	18	0	7
49	71	66	7	0

(5)  $k = 4$ 

0	22	17	35	42
22	0	5	13	20
17	5	0	18	25
35	13	18	0	7
42	20	25	7	0

(6)  $k = 5$ 

0	22	17	35	42
22	0	5	13	20
17	5	0	18	25
35	13	18	0	7
42	20	25	7	0

下面给出弗洛伊德算法的算法描述:

Procedure Floyd (G)

```

For i ← 1 to n do
  For j ← 1 to n do
     $A_{i,j} \leftarrow g_{i,j}$ 
    If  $a_{i,j} < \infty$  then  $p_{i,j} \leftarrow \{I\} \cup \{J\}$ 
    Else  $p_{i,j} \leftarrow \emptyset$ 
  For k ← 1 to n do
    For l ← 1 to n do
      For j ← 1 to n do
        If  $(I = k) \text{ or } (j = k) \text{ or } (I = j)$  then loop
        If  $a_{i,k} + a_{k,j} < a_{i,j}$  then
           $A_{i,j} \leftarrow a_{i,k} + a_{k,j}$ 
           $P_{i,j} \leftarrow p_{i,k} \cup p_{k,j}$ 
      Return (A, P)

```

### 例题 7-2 雅典奥运志愿者。

#### 问题描述：

虽然雅典在体育馆之间修筑了天桥，但来到这里看奥运的大多数游客还是喜欢走雅典的街道，他们总是想知道某两个体育馆之间的最短距离，于是有许多志愿者专门回答游客的问题，但由于他们也不可能把这些信息完全记下来，所以总是要向总部询问。如果你是总部的一名工作人员，你能胜任这个工作吗？

#### 输入：

第一行两个数  $n, m$  ( $1 \leq n \leq 100, 1 \leq m \leq 500$ )。表示一共有  $n$  个体育馆，有  $m$  条询问。

接下来一个  $n * n$  的矩阵，其中第  $I$  行第  $J$  列的元素表示体育馆  $I$  到体育馆  $J$  之间的距离，如果为 0，则表示没有直接的通路。

接下来有  $m$  行，每行两个数，表示询问的两个体育馆  $I, J$ 。

#### 输出：

对于每条询问输出一行，表示这两个体育馆之间的最短距离。

#### 分析：

本题是一道求最短距离的算法，可以对于每条询问用迪杰斯特拉算法求一次单源点的最短路径，也可以先用弗洛伊德算法求出任意两点之间的最短路径长度，然后对每条询问直接查找答案。下面分别给出两种算法的代码。

#### 源程序：

##### Dijkstra 算法：

```

const
  fin = 'input.txt';
  fou = 'output.txt';
  none = 15000;
  maxn = 100;
  maxm = 500;

```





```

var
  s: array [1..maxn] of integer;
  g: array [1..maxn, 1..maxn] of integer;
  query: array [1..maxn, 1..2] of integer;
  n, m: integer;

procedure init;
var i, j: integer;
begin
  assign (input, fin); reset (input);
  readln (n, m);
  for i := 1 to n do
    for j := 1 to n do
      read (g [i, j]);
  for i := 1 to m do
    readln (query [i, 1], query [i, 2]);
  close (input);
end;

procedure dijkstra (k: integer);
var i, j, t, ss: integer;
begin
  for j := 1 to n do begin
    if j <> k then s [j] := 0 else s [j] := 1;
    if g [j, k] = 0 then g [j, k] := none;
  end;
  for t := 1 to n - 2 do begin
    i := 0; ss := maxint;
    for j := 1 to n do
      if (s [j] = 0) and (g [k, j] < ss) then begin
        ss := g [k, j]; i := j;
      end;
    if i = 0 then exit;
    s [i] := 1;
    for j := 1 to n do
      if (s [j] = 0) and (g [k, i] + g [i, j] < g [k, j]) then
        g [k, j] := g [k, i] + g [i, j];
  end
end

```



```
end;

procedure main;
var k: integer;
begin
    for k := 1 to n do dijkstra (k)
end;

procedure print;
var i: integer;
begin
    assign (output, fou); rewrite (output);
    for i := 1 to n do
        writeln (g [query [i, 1], query [i, 2]]);
    close (output);
end;

begin
    init;
    main;
    print
end.
```

Floyd 算法:

```
const
    fin = 'input.txt';
    fon = 'output.txt';
    maxn = 100;
    maxm = 500;

var
    g: array [1..maxn, 1..maxn] of integer;
    query: array [1..maxm, 1..2] of integer;
    n, m: integer;

procedure init;
var
    i, j: integer;
begin
```



```
assign (input, fin); reset (input);
readln (n, m);
for i : = 1 to n do
  for j : = 1 to n do
    read (g [i, j]);
  for i : = 1 to m do
    readln (query [i, 1], query [i, 2]);
  close (input);
end;

procedure floyd;
var
  i, j, k : integer;
begin
  for k : = 1 to n do
    for i : = 1 to n do if (k <> i) and (g [i, k] > 0) then
      for j : = 1 to n do if (k <> j) and (i <> j) and (g [k, j] > 0) then
        if (g [i, j] = 0) or (g [i, k] + g [k, j] < g [i, j]) then
          g [i, j] : = g [i, k] + g [k, j];
      end;
    end;

  procedure print;
  var
    i : integer;
  begin
    assign (output, fon); rewrite (output);
    for i : = 1 to m do
      writeln (g [query [i, 1], query [i, 2]]);
    close (output);
  end;

begin
  init;
  floyd;
  print;
end.
```



## 7.7 拓扑排序

在实际工作中,经常用一个有向图来表示施工的流程图,或产品生产的流程图。一个工作往往可以分为若干个子工程,把子工程称为“活动”。在有向图中若以顶点表示“活动”的网(Activity On Vertex Network),简称为AOV网。

对于一个AOV网,构造其所有顶点的线性序列,使此序列不仅保持网中各顶点间原有的先后关系,而且使原来没有先后关系的顶点之间也建立起人为的先后关系。这样的线性序列称为拓扑有序序列。构造AOV网的拓扑有序序列的运算称为拓扑排序。

某个AOV网,如果它的拓扑有序序列被构造成功,则该网中不存在有向回路,其各个子工程可按拓扑有序序列的次序进行安排。显然,一个AOV网的拓扑有序序列并不是唯一的。图7-12所示的AOV网的拓扑有序序列:

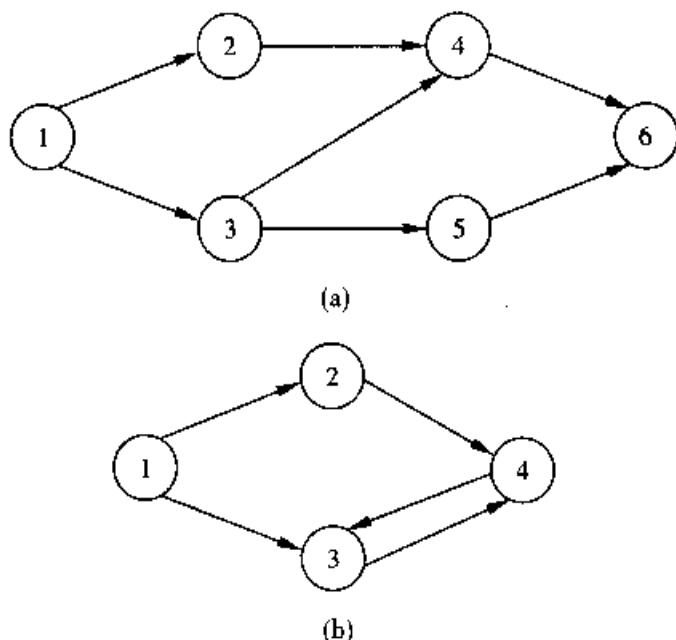


图7-12 AOV网示例

$V_1, V_2, V_3, V_4, V_5, V_6$

$V_1, V_3, V_2, V_5, V_4, V_6$

对AOV网进行拓扑排序的方法和步骤是:

- (1) 在网中选择一个没有前趋的顶点且输出之;
- (2) 从网中删去该顶点,并且删去从该顶点发出的全部有向边;
- (3) 重复上述两步,直至网中不存在没有前趋的顶点为止。

其操作的结果有两种:一是网中全部顶点均被输出,说明网中不存在有向回路,可以进行拓扑排序;另外是网中顶点未被全部输出,剩余的顶点均有前趋顶点,说明网中存在有向回路,不能够进行拓扑排序。图7-12(a)和(b)即是这两种情况。

在图7-12(a)中的AOV网,首先取没有前趋的顶点 $V_1$ 进行输出,并从网中删去 $V_1$ 及 $V_1$

发出的弧  $\langle V_1, V_2 \rangle$ ,  $\langle V_1, V_3 \rangle$ ; 接着, 再选没有前趋的顶点, 这时可选  $V_2, V_3$  中任意一个, 比如选  $V_2$ , 输出之, 删去  $V_2$  及弧  $\langle V_2, V_4 \rangle$ , 依此类推。最后将网中的顶点全部输出, 就得到了该网的一种拓扑排序序列:  $V_1, V_2, V_3, V_4, V_5, V_6$ 。

在图 7-12 (b) 中的 AOV 网, 首先取没有前趋的顶点  $V_1$  进行输出, 并从网中删去  $V_1$  及  $V_1$  发出的弧  $\langle V_1, V_2 \rangle$ ,  $\langle V_1, V_3 \rangle$ ; 接着, 再选没有前趋的顶点  $V_2$ , 输出之, 删去  $V_2$  及弧  $\langle V_2, V_4 \rangle$ ; 此时, 剩下的两个顶点中再也没有无前趋的顶点, 拓扑排序无法进行下去, 显然, 网中存在  $V_2, V_3$  一个有向回路。

为了在计算机上实现拓扑排序算法, AOV 网采用邻接表表示较方便, 不过要在表头结点结构中增加一个保存顶点入度的域 (indegree)。修改后的表头结点结构为:

data	indegree	link
------	----------	------

对于图 7-13 (a) 所示的 AOV 网, 它的邻接表如图 7-13 (b) 所示。每个顶点的入度值可以在建立邻接表或遍历邻接表时计算并填写。在进行拓扑排序中, 为了把所有入度为 0 的顶点都保存起来, 而且又便于插入、删除以及节省存储, 最好的方法是把它们链接成一个栈。另外, 当一个顶点的入度为 0 时, 该表头结点的入度域 (indegree) 就没有用了, 可正好作为链栈使用, 用来保存下一个入度为 0 的顶点的序号, 通过所有入度为 0 的表头结点的入度域就可以把入度为 0 的顶点 (对应表头结点) 都静态链接起来。在这个链栈中, 栈顶指针  $top$  指向第一个入度为 0 的顶点  $v_i$ ,  $v_i$  的表头结点的入度域指向第二个人度为 0 的顶点  $v_j$ , 依此类推, 最后一个人度为 0 的表头结点的入度域应置为 0, 表示栈底。

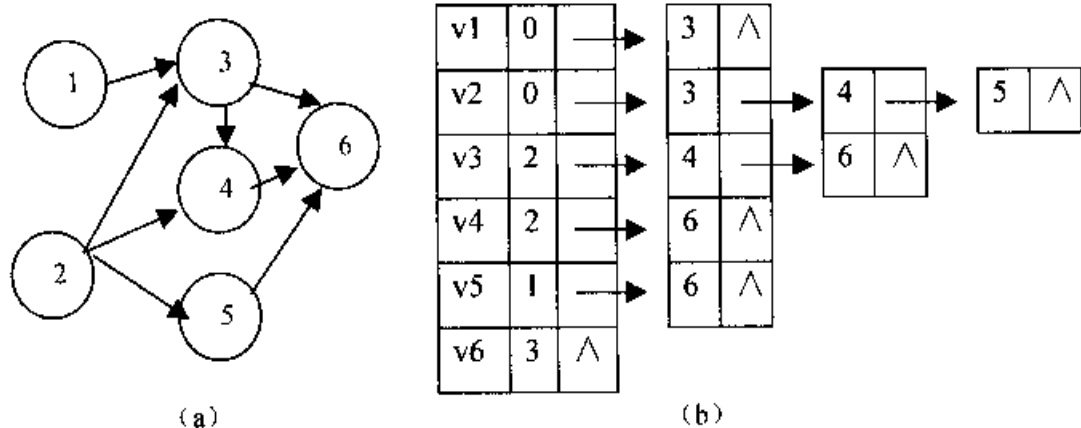


图 7-13 AOV 网及其邻接表图

例如, 根据图 7-13 (b) (假定它的表头向量为 GL), 建立的入度为 0 的初始栈的过程为:

(1) 开始置链栈为空, 即  $top: = 0$ 。

(2) 将入度为 0 的顶点  $v_1$  进栈, 即  $GL[1] \cdot indegree: = top; top: = 1$ ; 此时  $top$  指向第一个入度为 0 的顶点  $v_1$ ,  $v_1$  表头结点的入度域为 0, 表明到栈底。

(3) 将入度为 0 的顶点  $v_2$  进栈, 即  $GL[2] \cdot indegree: = top; top: = 2$ ; 此时  $top$  指向第一个入度为 0 的顶点  $v_2$ ,  $v_2$  表头结点的入度域为 1, 表明第 2 个人度为 0 的顶点为  $v_1$ ,  $v_1$  表头结点的

入度域为 0，所以此栈当前有两个顶点  $v_2$  和  $v_1$ 。

(4) 因  $v_3$  到  $v_6$  顶点的入度域均不为 0，所以它们均不进栈，至此，初始栈建立完毕。

将入度为 0 的顶点利用上述链栈链接起来后，拓扑算法中循环执行的第 (1) 步“选择一个入度为 0 的顶点并输出之”，可通过输出栈顶指针  $top$  所指向的顶点来处理；第 (2) 步“从网中删除刚输出的顶点（假定为  $v_j$ ）及所有出边”，可通过首先作退栈处理，使  $top$  指向下一个入度为 0 的顶点，然后遍历  $v_j$  的邻接表，分别把所有邻接点的入度减 1，若减 1 后的入度为 0 则令该顶点进栈来实现。此外，该循环的终止条件“直到不存在入度为 0 的顶点为止”，可通过判断栈空来实现。

根据以上分析，给出拓扑排序算法的具体描述为：

Procedure Toposort (GL); {对用邻接表 GL 表示的图进行拓扑排序}

Begin

Top: = 0; {将链栈置空}

For i: = 1 to n do {把所有初始入度为 0 的顶点序号进栈}

If GL [i]. indegree = 0 then [ GL [i]. indegree: = top; top: = i];

M: = 0; {用 m 记录输出顶点的个数}

While top < > 0 do

(1) j: = top; {j 的值为一个入度为 0 的顶点序号}

(2) top: = GL [top]. indegree; {退栈，使 top 指向下一个入度为 0 的顶点}

(3) write (GL [j]. data); {输出顶点  $v_j$  的值}

(4) m: = m + 1;

(5) p: = GL [j]. link; {p 指向  $v_j$  邻接表的第一个结点}

(6) while p < > nil do

(a) k: = p<sup>^</sup>. adjvex; {vk 是  $v_j$  的一个邻接点}

(b) GL [k]. indegree: = GL [k]. indegree - 1; 过 {vk 的入度减 1}

(c) If GL [k]. indegree = 0 then

[ GL [k]. indegree: = top; top: = k]; {把入度为 0 的顶点进栈}

(d) P: = p<sup>^</sup>. next; {p 指向  $v_j$  邻接表的下一个结点}

If m < n then writeln ( 'the network has a cycle' )

End;

若利用图 7-13 (b) 所示的邻接表来调用此算法，则得到的拓扑序列为：

$v_2, v_5, v_1, v_3, v_4, v_6$

拓扑实际上是对邻接表表示的图 G 进行遍历的过程，每次访问一个入度为 0 的顶点。若图 G 中没有回路，则需要扫描邻接表中的所有边结点，再加上在算法开始时，为建立入度为 0 的顶点的初始栈，对表头向量中的 n 个结点扫描一遍，所以，此算法的时间复杂性为  $O(n+e)$ 。

### 例题 7-3 士兵排队。

问题描述：

有 N 个士兵 ( $1 \leq N \leq 100$ )，编号依次为 1, 2, 3, ...。队列训练，指挥官要把一些士兵从高到矮一次排成行，但现在指挥官不能直接获得每个人的身高信息，只能获得“P1 比 P2 高”这样的比较结果 ( $P1, P2 \in 1, 2, 3, \dots, N$ )，如“A B”表示 A 比 B 高。

输入:

第一行为一个数  $N$  ( $N \leq 100$ ), 表示士兵的个数。以下若干行, 每行有两个数  $A, B$ , 表示士兵  $A$  的身高大于士兵  $B$  的身高。

输出:

给出一个合法的排队序列。

分析:

此题初看, 不知道如何下手。因此不妨先用图将上述身高关系表示出来。图中有  $N$  个点分别代表  $N$  个士兵。若已知士兵  $A$  的身高大于士兵  $B$  的身高, 则在  $A, B$  间连一条从  $A$  到  $B$  的有向边。这时发现, 原问题就是要求此图的一个拓扑排序。如图 7-14 所示的一个实例,  $A$  比  $B$  高,  $B$  比  $C$  高,  $D$  比  $C$  高。一个可能的士兵排队顺序是  $ABDC$ 。

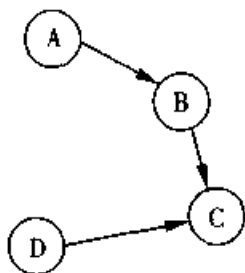


图 7-14 示例图

源程序:

```

const
  fin = 'input.txt';
  fon = 'output.txt';
  maxn = 100;
var
  map : array [1..maxn, 1..maxn] of integer; {记录有向边的情况}
  id,                                     {记录每个点的入度}
  top : array [1..maxn] of integer;         {记录拓扑序列}
  n : integer;

procedure init;
var
  i, j : integer;
begin
  assign (input, fin); reset (input);
  readln (n);
  while not seekeof do begin
    readln (i, j);
    map [i, j] := 1;
    inc (id [j]);
  end;

```

```

    end;
    close (input);
end;

procedure print (ans : integer);
var
    i : integer;
begin
    assign (output, fon); rewrite (output);
    if ans = - 1 then writeln ( 'No solution. ' )
    else
        for i : = 1 to n do
            writeln (top [i]);
        close (output);
    halt;
end;

procedure main;
var
    i, j, k : integer;
begin
    for i : = 1 to n do begin
        j : = 1;
        while (j ≤ n) and (id [j] < > 0) do inc (j);
        if j > n then print ( - 1);
        top [i] : = j; id [j] : = maxint;
        for k : = 1 to n do
            if map [j, k] = 1 then dec (id [k]);
        end;
    end;
end;

begin
    init;
    main;
    print (1);
end.

```



## 7.8 关键路径

若在带权有向图  $G$  中, 以顶点表示事件, 以有向边表示活动, 边上的权值表示该活动持续的时间, 则此带权有向图称为用边表示“活动”的网 (Activity On Edge Network), 简称 AOE 网。通常在 AOE 网上列出了完成预定工程计划所需要进行的活动、每项活动的计划完成时间、要发生哪些事件及这些事件和活动间的关系, 从而可以分析该项工程是否实际可行, 估算工程的完成时间, 哪些活动是影响工程进度的关键, 进一步可以进行人力、物力的调度和分配以达到缩短工期的目的。

在 AOE 网表示一项工程的施工计划时, 顶点所表示的事件实际上就是某些活动已经完成以及某些子工程可以动工的标志。具体地说, 顶点所表示的事件是指该顶点所有进入边所表示的活动均已完成以及它的发出边所表示的活动均可以开始的一种状态。例如, 图 7-15 是一个具有 4 项活动的假想工程的 AOE 网。网中共有 4 个顶点, 分别表示 4 个事件。边上的权值表示要完成该边所表示的活动所需要的时间, 如图 7-15 中活动  $a_1$  计划需花 3 天时间才能完成。关系这个 AOE 网的各个事件的含义解释如下: 事件  $v_1$ , 表示该工程的开始; 事件  $v_2$ , 表示活动  $a_1$  完成而活动  $a_3$  可以开始; 事件  $v_3$ , 表示活动  $a_2$  完成而活动  $a_4$  可以开始; 时间  $v_6$  表示该工程结束。

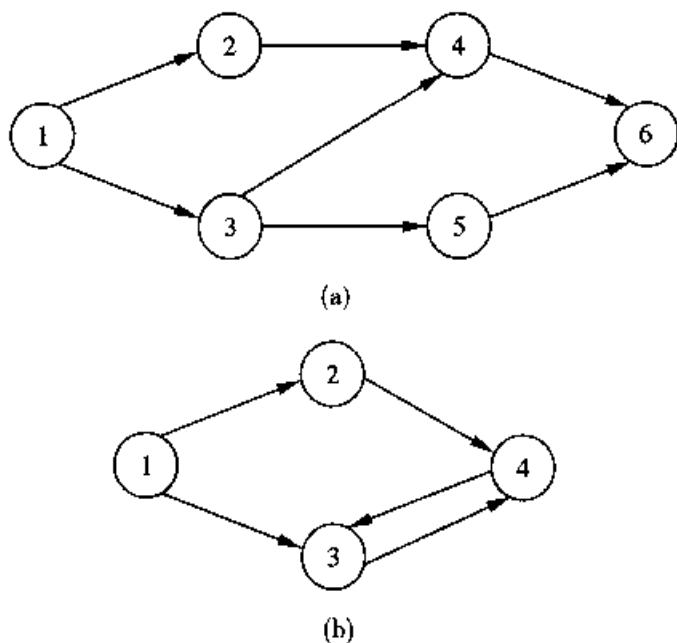


图 7-15 AOE 网示意图

对于一个工程来说, 一般有一个开始状态和一个结束状态, 所以在 AOE 网中至少有一个开始顶点。开始顶点的入度为零亦称为源点; 另外有一个结束顶点, 结束顶点的出度为零, 亦称为汇点。网中不能存在有向回路, 否则整个工程无法完成。

与 AOV 网不同, 对于 AOE 网所关心的是: 完成该工程至少需要多少时间以及哪些活动是影响整个工程进度的关键。

由于 AOE 网中的某些活动能够平行地进行, 故完成整个工程所需的时间是从开始顶点到结束

顶点的最长路径长度。这里的路径长度是指该路径上的权值之和。从源点到汇点的路径长度最长的路径称为关键路径。

关键路径上的所有活动均是关键活动。如果任何一项关键活动没有按期完成,则要影响整个工程的进度,而提高关键活动的速度通常可以缩短整个工程的工期。例如图 7-15 的 AOE 网,关键路径是  $v_1-v_3-v_4$ , 关键路径长度为  $2+4=6$ , 关键活动是  $a_2, a_4$ 。如果提高  $a_4$  的速度,使其由原计划的 4 天减少到 3 天,则整个工程可以提前一天完工。若进一步提高  $a_4$  的速度,使其只用 2 天完成,但整个工程不能在 4 天内完成。这是因为当  $a_4=2$  时,关键路径就变化了。如果一个 AOE 网中存在两条以上的关键路径,则需同时提高这几条关键路径上某些关键活动的速度,才能缩短整个工程的工期。如果提高非关键活动的速度是不能加快整个工程进度的。

由上分析可知,要对工程计划进行有效的安排和调度,首先应求出其对应的 AOE 网的关键路径和关键活动。为了求出关键活动,先定义几个有关的变量,并讨论其计算方法。

(1) 顶点事件的最早发生时间  $Ve(j)$ 。 $Ve(j)$  是指从源点  $V_1$  到  $V_j$  的最长路径长度。这个时间决定了所有  $V_j$  发出的弧所表示的活动能够开工的最早日期。

$Ve(j)$  的计算方法为:

$$Ve(1) = 0$$

$$Ve(j) = \max \{Ve(i) + dut\langle i, j \rangle\} \quad \langle i, j \rangle \in T, 2 \leq j \leq n$$

其中  $T$  是所有到达顶点  $j$  的弧的集合,  $dut\langle i, j \rangle$  是弧  $\langle i, j \rangle$  上的权值;  $n$  是网中的顶点数。

显然,上述公式是一个从源点开始的递推公式。 $Ve(j)$  的计算必须在  $V_j$  的所有前趋顶点的最早发生时间全部求出后才能进行。这样必须对 AOE 网进行拓扑排序,然后按拓扑有序序列逐个求出各顶点事件的最早发生时间。例如图 7-14 中的顶点  $V_4$ ,其最早发生时间  $Ve(4)$  是在求得  $Ve(2)=3, Ve(3)=2$  之后,才求得  $Ve(4)=6$ 。

(2) 顶点事件的最晚发生时间  $Vi(I)$ 。 $Vi(I)$  是指在不推迟整个工程完成日期的前提下,事件  $Vi$  所允许的最晚发生时间。对一个工程来说,计划用几天时间完成是可以从 AOE 网中求得的,其数值就是汇点  $Vn$  的最早发生时间  $Ve(n)$ ,而这个时间也就是  $Vi(n)$ 。其他顶点事件的  $Vi$  应从汇点开始,逐步向源点方向递推才能求出。

$Vi$  的计算公式应该是:

$$Vi(n) = Ve(n)$$

$$Vi(i) = \min \{Vi(j) - dut\langle i, j \rangle\} \quad \langle i, j \rangle \in S, 1 \leq i \leq n-1$$

其中,  $S$  是所有从顶点  $i$  发出的弧的集合。

显然,  $Vi(I)$  的计算必须在顶点  $I$  的所有后继顶点的最晚发生时间全部求出后才能进行。这样必须对 AOE 网进行逆拓扑排序,然后按逆拓扑有序序列进行递推求出各顶点事件的  $Vi$ 。例如图 7-13 中的  $V_2$ ,其  $Vi(2)$  是在先求出  $Vi(4)=6$  之后才求得  $Vi(2)=4$ 。

(3) 边活动的最早开始时间  $Ee(I)$ 。 $Ee(I)$  是指该边所表示的活动  $ai$  最早可以开工的时间。若活动  $ai$  是由弧  $\langle j, k \rangle$  表示,则:  $Ee(I) = Ve(j)$ 。这说明活动  $ai$  的最早开始时间等于事件  $Vj$  的最早开始时间。这与 AOE 网中关于事件含义的解释是完全一致的。

(4) 边活动的最晚开始时间  $Ei(I)$ 。 $Ei(I)$  是指在不推迟整个工程完成日期的前提下,允许该活动最晚开始的时间。若活动  $ai$  由弧  $\langle j, k \rangle$  表示,则  $Ei(I) = Vi(k) - dut\langle j, k \rangle$ 。

对于活动  $ai$  来说,若  $Ee(I) = Ei(I)$ ,表明该活动最早可以开工的日期与整个工程计划允许该活动最迟的开工日期相等,施工时间一点也不能拖延。若  $ai$  活动不能按计划日期完成,则

整个工程就要延期。若它提前完成,则有可能使整个工程也提前完成,该活动就是关键活动。

因此,对于活动  $a_i$ ,若  $E_i(I) - E_e(I) = 0$ ,则  $a_i$  是关键活动。由关键活动组成的路径就是关键路径。现在来讨论关键路径的具体算法:

先从源点出发进行拓扑排序,它的作用是判断是否存在回路和可以进行拓扑时求出每个事件的最早发生时间  $V_e$ ;然后从汇点出发进行逆拓扑排序,在逆拓扑排序过程中求出每个事件的最晚发生时间  $V_l$ ;第三步根据上面求出  $V_e$  和  $V_l$ ,求出每个活动最早开始时间  $E_e$  和最晚开始时间  $E_l$ ;第四步求出关键路径,根据第三步求出的每个活动的最早开始时间  $E_e$  和最晚开始时间  $E_l$ ,如果有  $E_e(j) = E_l(j)$ ,则  $j$  活动为关键活动。

下面介绍在竞赛中常用的一种较好的方法:

(1) 从源点出发进行拓扑排序,若拓扑排序无法进行下去,则说明网中存在有向回路,因此不能求出关键路径。这里拓扑排序起到了用动态规划求解的划分阶段的作用。

(2) 用动态规划求解关键路径。

#### 例题 7-4 工程问题。

问题描述:

在大型工程的施工前,经常把整个工程划分为若干个子工程,并把这些子工程编号为 1, 2, ..., N。这样划分之后,子工程之间就会有一些依赖关系,即一些子工程必须在某些子工程之后才能施工。由于子工程之间有相互依赖关系,因此有两个任务需要我们去完成:首先,需根据每个子工程的完成时间计算整个工程最少的完成时间;另一方面,由于一些不可预测的客观因素使某些子工程延期,因此必须知道哪些子工程的延期会影响整个工程的延期,我们把有这种特征子工程称为关键子工程。因此第二个任务就是找出所有的关键子工程,以便集中精力管理好这些子工程,尽量避免这些子工程延期,达到用最快的速度完成整个工程的目的。

(1) 根据预算,每个子工程都有一个完成时间;

(2) 子工程之间的依赖关系是:部分子工程必须在一些子工程完成之后才能开工;

(3) 只要满足子工程的依赖关系,在任何时刻可以有任何多个子工程同时在施工,即同时施工的子工程个数不受限制;

(4) 整个工程的完成是指所有子工程完成。

输入:

第 1 行为 N, N 是子工程的总个数,  $N \leq 100$ ; 第 2 行为 N 个正整数,分别代表 N 个子工程 1, 2, ..., N 的完成时间;第 3 行到 N+2 行,每行有 N-1 个 0 或 1。其中的第 I+2 行的这些 0, 1 分别代表“子工程 I”与子工程 1, 2, ..., I-1, I+1, ..., N 的依赖关系 ( $I=1, 2, \dots, N$ )。

输出:

如子工程划分不合理,则输出 -1;如子工程划分合理,则用两行输出,第 1 行为整个工程最少的完成时间,第 2 行为按由小到大顺序输出所有关键子工程的编号。

样例:

输入:          输出:

```
Input.txt  Output.txt
5          14
5 4 12 7 2 1 3 4 5
0 0 0 0
```

0 0 0 0

0 0 0 0

1 1 0 0

1 1 1 1

分析:

本题从题目来看与关键路径问题如出一辙,但是也稍微有一些变化,即边上无权,但点上有权。不过这点改动并不影响问题的解决,仍然可以采用与关键路径相类似的算法。具体实现时,由于有多个无前趋和多个无后继的结点,再进行递推时不方便,所以可以在图中设计一个虚拟头结点和一个虚拟尾结点。从虚拟头结点向每一个工程引出一条有向边,从每一个工程引出一条有向边到虚拟尾结点。

下面给出源程序:

```
const
  fin = 'input.txt';
  fon = 'output.txt';
  maxn = 100;

var
  map : array [0..maxn + 1, 0..maxn + 1] of integer; {记录工程之间依赖关系}
  times,                                           {记录每个工程的耗时}
  ve,                                           {记录顶点事件最早发生时间}
  vl,                                           {记录顶点时间最晚发生时间}
  top,                                           {记录拓扑序列}
  id : array [0..maxn + 1] of integer;          {记录每个顶点的入度}
  n : integer;

procedure init;
var
  i, j : integer;
begin
  assign (input, fin); reset (input);
  readln (n);
  for i : = 1 to n do read (times [i]);
  for i : = 1 to n do begin
    for j : = 1 to n do
      if i < > j then
        read (map [j, i]);
    map [0, i] : = 1;
    map [i, n + 1] : = 1;
  end;
  close (input);
```



```

end;

procedure prepare;
var
    i, j : integer;
begin
    for i : = 0 to n + 1 do
        for j : = 0 to n + 1 do
            inc (id [j], map [i, j]);
        end;
    end;

function topsort : boolean;
var
    i, j, k : integer;
begin
    topsort : = false;
    for i : = 0 to n + 1 do begin
        j : = 0;
        while (j ≤ n + 1) and (id [j] < > 0) do inc (j);
        if j > n + 1 then exit;
        top [i] : = j;
        id [j] : = maxint;
        for k : = 0 to n + 1 do
            if map [j, k] = 1 then dec (id [k]);
        end;
        topsort : = true;
    end;

procedure calve;
var
    i, j, x, y : integer;
begin
    ve [0] : = 0;
    for i : = 1 to n + 1 do begin
        x : = top [i];
        for j : = 0 to i - 1 do begin
            y : = top [j];
            if (map [y, x] = 1) and (ve [y] > ve [x]) then
                ve [x] : = ve [y];
        end;
    end;
end;

```



```

end;
ve [x] := ve [x] + times [x];
end;
end;

procedure calcvl;
var
    i, j, x, y : integer;
begin
    vl [n + 1] := ve [n + 1];
    for i := n downto 0 do begin
        x := top [i];
        vl [x] := maxint;
        for j := i + 1 to n + 1 do begin
            y := top [j];
            if (map [x, y] = 1) and (vl [y] - times [y] < vl [x]) then
                vl [x] := vl [y] - times [y];
        end;
    end;
end;

procedure print (ans : integer);
var
    i : integer;
begin
    assign (output, fon); rewrite (output);
    if ans = - 1 then writeln ('No solution')
    else begin
        writeln (ve [n + 1]);
        for i := 1 to n do
            if ve [i] = vl [i] then write (i, ' ');
        writeln;
    end;
    close (output);
    halt;
end;

procedure main;
begin

```

```

prepare;
if not topsort then print (-1);
calcev;
calcvl;
end;

begin
  init;
  main;
  print (1);
end.

```

## 7.9 图的应用举例

图是一种相对较复杂而且很重要的数据结构，应用相当广泛。下面通过几个实例，谈谈图在信息学中的具体应用和技巧。

### 例题 7-5 马拉松赛跑。

问题描述：

长郡中学想在百年校庆的那天举行一个万人马拉松赛跑，凡是长郡中学毕业的学生，都可以参加。校内肯定是没有这么大的地方，所以向市政府申请了一些道路，在这些道路上进行马拉松赛跑。为了不给交通太大的压力，所以道路都只占用了一半，为了安全起见，这样的道路都作为单向的。可以在任何道路的任何位置开始及结束。考虑到参加比赛的毕竟都不是专业运动员，赛程不可能太长，但不知道对于某个长度给定的赛场是否能够胜任，这个问题就交给你了。

输入：

第一行有三个数  $n, m, s$ ，分别表示道路的交叉路口数目、道路数目以及给定的赛程长度。

接下来  $m$  行，每行三个数  $p, q, r$  表示在第  $p$  个交叉路口到第  $q$  个交叉路口之间有一条长度为  $r$  的道路。

输出：

如果赛场能够找出一条给定长度的赛道则输出 YES，否则输出 NO。

样例：

输入	输出
3 2 20	NO
1 2 10	
2 3 5	

分析:

该题实际上就是问图中是否存在一条长度大于等于  $s$  的路径, 而且该路径可以重复经过一个顶点多次。如果图中有一个圈, 那么不管  $s$  为多大, 一定能找到。如果图中没有圈, 那么每个连通分量都是一棵树, 只需要在树中找一条最长路径, 看是否大于等于  $s$  即可, 注意要处理每一个连通分量。

源程序:

```
Const fin = 'input.txt';
      fon = 'output.txt';
      MaxN = 101;
Var   map : array [1..MaxN, 1..MaxN] of integer;
      visited : array [1..MaxN] of boolean;
      fa : array [1..MaxN] of integer;
      n, m : integer;
      l, longest : longint;

Procedure Print (ans : integer);
begin
  assign (output, fon); rewrite (output);
  if ans = 1 then writeln ('YES') else writeln ('NO');
  close (output);
  halt;
end;

Procedure Init;
var i, x, y, r : integer;
begin
  assign (input, fin); reset (input);
  readln (n, m, l);
  for i := 1 to m do begin
    readln (x, y, r);
    if x = y then Print (1);
    if map [x, y] = 0 then begin
      map [x, y] := r; map [y, x] := r;
    end else Print (1);
  end;
  close (input);
end;
```



```

Procedure Search (i : integer);
var j : integer;
begin
  for j : = 1 to n do
    if (map [i, j] > 0) and (j < > fa [i]) then begin
      if fa [j] < > 0 then Print (1);
      fa [j] : = i;
      Search (j);
    end;
  end;
end;

Procedure Calc (i : integer; var first : longint);
var j : integer;
    second, temp : longint;
begin
  first : = 0; second : = 0;
  for j : = 1 to n do if fa [j] = i then begin
    Calc (j, temp); temp : = temp + map [i, j];
    if temp > first then begin second : = first; first : = temp end
    else if temp > second then second : = temp;
  end;
  if first + second > longest then longest : = first + second;
end;

Procedure Main;
var i, j : integer;
    temp : longint;
begin
  for i : = 1 to n do
    if fa [i] = 0 then begin
      fa [i] : = - 1;
      Search (I);
    end;
  for i : = 1 to n do
    if fa [i] = - 1 then begin
      Calc (i, temp);
      if temp > longest then longest : = temp;
    end;
  end;
end;

```

```

if longest ≥ 1 then Print (1) else Print (0);
end;

Begin
    Init;
    Main;
End.

```

### 例题 7-6 沙丘。

#### 问题描述：

根据新出土的一批史料记载，在塔克拉玛干沙漠中的一座沙丘下面，埋藏着一个神秘的地下迷宫。由著名探险家阿强率领的探险队经过不懈的挖掘，终于发现了通往地下迷宫的入口！队员们兴奋不已，急忙钻下去，去寻找那个埋藏已久的秘密。

他们刚钻进迷宫，只听“轰隆”一声巨响，回头一看，入口已与石墙融为一体，无法辨认。他们意识到自己被困在迷宫里了！环顾周围，似乎是一个洞穴。

这座迷宫由很多洞穴组成，某些洞穴之间有道路连接。每个洞穴都有一盏灯，凭借着微弱的灯光，可以看清有多少条道路与这个洞穴相连。每个洞穴的内部是完全相同的，且无法做标记。每条道路也是完全相同的，也无法做标记。

阿强凭借着微弱的灯光，发现了墙壁上的一段文字（事实上，每个洞穴的墙壁上都有这段文字），翻译成现代汉语就是：“陌生人，请把这个迷宫的洞穴数和道路数告诉我，我就会指引你走出迷宫。”

阿强很快镇定了下来，他拿出一个路标，对队员们说：“这个迷宫的危险程度远超出我们的想象，为了安全起见，大家一定要集体行动。我这儿有一个路标，有了它，我们一定能探明迷宫的结构。大家跟我走！”

现在，轮到你扮演阿强了。路标只有一个，可以随身携带，也可以暂时放在某个洞穴中（把路标放在道路上是毫无意义的，因为那里一片漆黑，什么都看不见）。你的任务很简单：用尽量少的步数探明这个迷宫共有多少个洞穴和多少条道路。“一步”是指从一个洞穴走到另一个相邻的洞穴。

#### 交互方法：

本题是一道交互式题目，你的程序应当和测试库进行交互，而不得访问任何文件（包括临时文件）。测试库提供了若干函数，它们的用法和作用如下：

- \* `init` 必须先调用，但只能调用一次，用作初始化测试库；
- \* `look (d, sign)` 的作用是查看当前洞穴的情况，测试库将从整型变量 `d` 中返回与该洞穴相连的道路的数目，从布尔型变量 `sign` 中返回该洞穴内是否有路标，`sign` 为 `true` 表示有路标，为 `false` 表示无路标。
- \* `pulsign` 的作用是在当前洞穴放上路标。只有当路标随身携带着的时候，才可以调用这个函数。
- \* `takesign` 的作用是把当前洞穴的路标拿走。只有当路标在当前洞穴时，才可以调用这个函数。
- \* `walk (i)` 的作用是沿着编号为 `i` 的道路走到相邻的洞穴中。这里的编号是相对于当前所

在洞穴而言的，并且是暂时的。假设与某洞穴相连的道路有  $d$  条，这些道路按照逆时针顺序依次编号为  $0, 1, 2, \dots, d-1$ 。走第一步时，编号为  $0$  的道路由库确定。以后的过程，阿强会将他走进这个洞穴的道路编号为  $0$ 。

\* `report (n, m)` 的作用是向测试库报告结果。 $n$  表示洞穴的数目， $m$  表示道路的数目。当这个函数被调用后，测试库会自动中止你的程序。

样例：

dune.in 内容如下

```
5
3 2 3 4
2 1 3
2 1 2
2 1 5
1 4
```

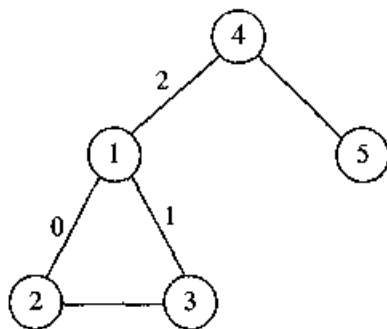


图 7-16 示例图

探险队初始时站在编号为  $1$  的洞穴内，编号为  $0$  的道路通向洞穴  $2$ ，编号为  $1$  的道路通向洞穴  $3$ ，编号为  $2$  的道路通向洞穴  $4$ 。

一种可能得满分的调用方案如下：

Pascal 选手的调用方法	C/C++ 选手的调用方法	说 明
<code>init;</code>	<code>init ();</code>	初始化程序
<code>look (d, sign);</code>	<code>look (d, sign);</code>	返回 $d = 3$ , $sign = false$
<code>putsign;</code>	<code>putsign ();</code>	放下路标
<code>walk (0);</code>	<code>walk (0);</code>	选择编号为 $0$ 的道路
<code>look (d, sign);</code>	<code>look (d, sign);</code>	返回 $d = 2$ , $sign = false$
<code>walk (1);</code>	<code>walk (1);</code>	选择编号为 $1$ 的道路
<code>look (d, sign);</code>	<code>look (d, sign);</code>	返回 $d = 2$ , $sign = false$
<code>walk (1);</code>	<code>walk (1);</code>	选择编号为 $1$ 的道路
<code>look (d, sign);</code>	<code>look (d, sign);</code>	返回 $d = 3$ , $sign = true$

Pascal 选手的调用方法	C/C++ 选手的调用方法	说 明
takesign;	takesign ();	拿起路标
walk (1);	walk (1);	选择编号为 1 的道路
look (d, sign);	look (d, sign);	返回 $d = 2$ , $sign = false$
walk (1);	walk (1);	选择编号为 1 的道路
look (d, sign);	look (d, sign);	返回 $d = 1$ , $sign = false$
report (5, 5);	report (5, 5);	返回洞穴数为 5, 道路数为 5

分析:

初看这道题目会觉得无从下手, 因为对这个图的遍历显得很盲目。某个顶点或者某条边到底是否计算过呢? 标志只有一个, 应该怎样合理运用呢? 等等这样的问题都使得我们难以找到突破口。

不妨先确定一条很简单又显而易见的思路: 通过对图的遍历, 每走到一个新的结点或者新的边就统计, 如果重复则忽略。这样目标变得简单明了, 就可以按着这个思路来设计算法。

我们往往把图的问题利用生成树等具有更多特殊性质的数据结构来解决, 这道题目同样可以用这样的方法。对于任意一个图, 用深度优先遍历求得一棵生成树, 它把图中的边分成两类: 树边 ( $n - 1$  条), 非树边 ( $m - n + 1$  条), 而且所有的非树边都是后向边 (即某个点和它的祖辈结点之间的边), 而没有横向边, 这是很显然的, 但确是极有用的。有了这个性质, 也就是如果按照深度优先顺序遍历, 遇到了重复点必定是这个结点的祖辈结点。深度优先遍历时保存从根到当前结点的路径, 也就是生成树上该结点的所有祖辈结点。

假设当前遍历的结点为 I, 那么需要依次遍历除 I 的父结点之外的其他所有邻结点。如果当前遍历邻结点 J, 则首先把 sign 放在 J, 然后按照树上的边返回直到根或者在路径上发现 sign。

a) 如果在路径上发现 sign, 则表示 J 结点是 I 结点的一个祖辈结点, 那么统计边 (I, J), 并把从 J 到 I 的那条边做标记, 继续遍历 I 的下一个结点, 当回溯到 J 的时候不要再遍历 I 结点了。

b) 如果一直返回到根都没发现 sign, 则 J 是一个新结点, 那么统计边 (I, J) 和结点 J 并扩展顶点 J。

这样访问每条边都至少需要遍历一次到根的路径, 所以时间复杂度和询问次数的复杂度都是  $O(n * m)$ 。

接下来的问题是: 由于一个点的邻结点的编号时刻在发生变化, 怎样确定邻结点的编号? 设: Aroundi——表示点 I 把它父结点在它的邻结点中编号为 0 时, 它当前的子结点的编号是多少。

这样在处理的时候就会方便一些, 每次回溯的时候先回到 I 的父结点, 然后再 walk (0) 走到 1, 这样 I 的父结点在它的邻结点中编号为 0, 和 Aroundi 保持一致。

如图 7-17, A、B、C、D 是当前路径上的点, 而 E、F、G 是访问过的结点, H、I 是没有访

问的结点。而此时  $\text{Around}_A$  是 B,  $\text{Around}_B$  是 C,  $\text{Around}_C$  是 D。每访问该结点下一个点时, 需要修改  $\text{Around}$ 。

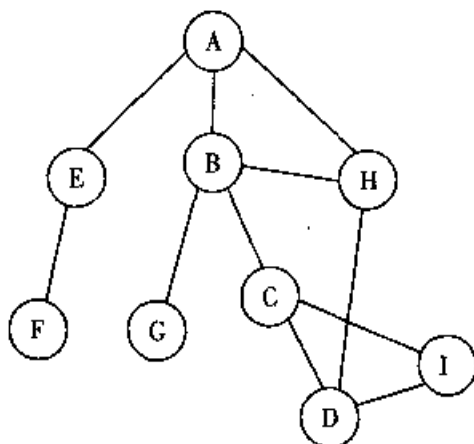


图 7-17 示例图

$\text{Visited}_{i,j}$ ——表示点 I 把它父结点在它的邻结点中编号为 0 时, 编号为 J 的结点是否遍历过了。

如图 7-18, 如果当前路径为 F, I, A, B, C, J, 当访问 J 的邻结点 I 时发现 I 是一个已经访问过的结点, 但是现在不知道 J 在 I 的邻结点中编号为多少, 所以需要求出 J 和 A 的编号的相对关系。那么把 Sign 放到 J 上, 然后顺着 C, B, A 返回到 I, 切记此时不能从 J 直接到 I, 因为这样会丢失当前的相对位置关系, 即 F 的位置。然后再依次枚举 I 的顺着 A 的下一个结点, 即 D, E, ... J, 直到发现 Sign 就可以得知 J 是 I 的哪一个邻结点了。

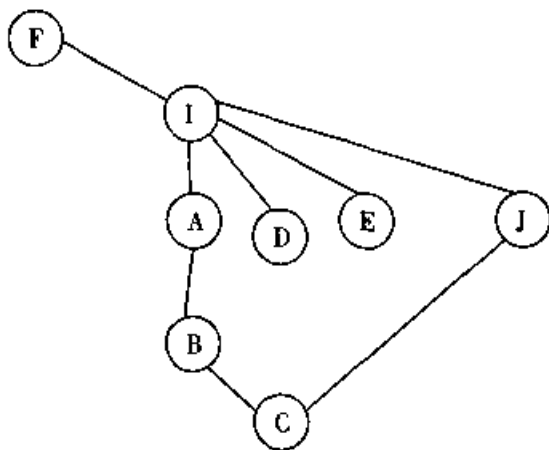


图 7-18 示例图

源程序:

```
uses dunelib;
const
    maxn = 101;
    maxm = 4001;
type integer = longint;
var
```

```
visited : array [1..maxn, 0..maxm] of boolean;
around : array [1..maxn] of integer;
n, m : integer;
```

```
procedure search (now : integer);
var i, j, k, t, d, dj : integer;
    sign, find : boolean;
begin
    inc (n);
    look (d, sign);
    fillchar (visited [now], sizeof (visited [now]), 0); around [now] := 0;
    for i := ord (now < > 1) to d - 1 do
        if not visited [now, i] then begin
            inc (m); visited [now, i] := true; around [now] := i;
            walk (1); putsign; walk (0);
            j := now; dj := d; find := false;
            while j > 1 do begin
                walk ((dj - around [j]) mod dj); dec (j);
                look (dj, sign);
                if sign then begin find := true; break; end;
            end;
            if not find then begin
                walk (0); inc (j);
                look (dj, sign);
                while not sign do begin
                    walk (around [j]); inc (j);
                    look (dj, sign);
                end;
                takesign;
                search (now + 1);
                walk ((d - around [now]) mod d);
                walk (0);
            end else begin
                k := j;
                walk (0); inc (j);
                look (dj, sign);
                while not sign do begin
                    walk (around [j]); inc (j);
                    look (dj, sign);
```

```

end;
takesign; walk (0); dec (j); putsign;
look (dj, sign);
while j > k do begin
    walk ((dj - around [j]) mod dj); dec (j);
    look (dj, sign);
end;
walk (1); look (dj, sign); t := 1;
while not sign do begin
    inc (t); walk (0);
    walk (1); look (dj, sign);
end;
takesign;
visited [k, around [k] + t] := true; walk (0);
look (dj, sign); walk ((dj - t) mod dj);
for j := k + 1 to now - 1 do walk (around [j]);
end;
end;
walk (0);
end;

begin
    init;
    search (1);
    report (n, m);
end.

```

可以发现, 这道题目的算法几乎没有用到什么高深的知识, 仅仅是以最基础的图的深度优先遍历为框架, 经过加工后得到的解法。可见, 掌握基础算法是很重要的, 许多看似高深的算法其实也就是—些最基础最简单的算法经过—定的变形加工后得到的。

#### 例题 7-7 出纳员的雇佣。

问题描述:

Tehran 的一家每天 24 小时营业的超市, 需要一批出纳员来满足它的需要。超市经理雇佣你来帮他解决他的问题——超市在每天的不同时段需要不同数目的出纳员 (例如: 午夜时只需一小批, 而下午则需要很多) 来为顾客提供优质服务。他希望雇佣最少数目的出纳员。

经理已经提供你一天的每一小时需要出纳员的最少数目—— $R_0, R_1, \dots, R_{23}$ 。 $R_0$  表示从午夜到上午 1:00 需要出纳员的最少数目,  $R_1$  表示上午 1:00 到 2:00 之间需要的, 等等。每一天, 这些数据都是相同的。有  $N$  人申请这项工作, 每个申请者  $I$  在 24 小时中, 从一个特定的时刻开始连续工作恰好 8 小时, 定义  $t_i$  ( $0 \leq t_i \leq 23$ ) 为上面提到的开始时刻。也就是说, 如果第  $I$  个申请者被录取, 他 (她) 将从  $t_i$  时刻开始连续工作 8 小时。你将编写一个程序, 输入  $R_i$  ( $i = 0..23$ ) 和

$t_i$  ( $i = 1..N$ ), 它们都是非负整数, 计算为满足上述限制需要雇佣的最少出纳员数目。在每一时刻可以有比对应的  $R_i$  更多的出纳员在工作。

输入:

输入文件的第一行为 24 个整数表示  $R_0, R_1, \dots, R_{23}$  ( $R_i \leq 1000$ )。接下来一行是  $N$ , 表示申请者数目 ( $0 \leq N \leq 1000$ ), 接下来每行包含一个整数  $t_i$  ( $0 \leq t_i \leq 23$ )。

输出:

对于每个测试点, 输出只有一行, 包含一个整数, 表示需要出纳员的最少数目。如果无解, 你应当输出 “No Solution”。

分析:

初看本题, 很容易使人往贪心、动态规划或网络流这些方面思考, 但这些算法对于本题都无能为力。由于本题的约束条件很多, 为了理清思路, 我们先把题目中的约束条件用数学语言表达出来。设  $S_i$  表示  $0 \sim i$  时刻雇佣出纳员的总数,  $W_i$  表示在时刻  $i$  开始工作的申请者的人数。那么我们可以将题目中的约束条件转化为下面的不等式组:

$$\begin{cases} 0 \leq S_i - S_{i-1} \leq W_i & 0 \leq i \leq 23 \\ S_i - S_{i-8} \leq R_i & 8 \leq i \leq 23 \\ S_{23} + S_i - S_{i+16} \leq R_i & 0 \leq i \leq 7 \end{cases}$$

这样的不等式组, 不禁使我们想到了差分约束系统。对于每条不等式  $S_i - S_j \leq K$ , 从顶点  $j$  向顶点  $i$  引一条权值为  $K$  的有向边。我们要求的  $S_{23}$  的最小值, 只要求顶点 0 到顶点 23 的最短路。但是注意上面第三条不等式: 它包含一个未知数, 无法在图中表示为边的关系。

思考到这一步, 似乎陷入了僵局。难道本题不能用差分约束系统解决吗? 不, 我们还需要一些转化。退一步海阔天空, 如果把  $S_{23}$  作为未知数, 那是肯定做不下去的。但是如果把  $S_{23}$  作为已知数, 那么第三条不等式就只有两个未知数  $S_i, S_{i+16}$ , 我们从顶点  $i+16$  向顶点  $i$  ( $0 \leq i \leq 7$ ) 引一条权值为  $R_i - S_{23}$  的边。

那么, 该不等式组可以完全转化为一个有向图, 未知数  $S_i$  的解, 就是图中顶点 0 到顶点  $i$  的最短路。而当图中存在负权回路时, 不等式组无解。上面的解法是把  $S_{23}$  当成了已知数, 而实际上  $S_{23}$  不但是未知的, 而且正是我们要求的。怎么办? 我们可以用二分法枚举  $S_{23}$  的值, 逐步缩小范围, 用迭代法判断是否存在负权回路 (判定可行性)。如果当  $S_{23}$  取到  $N$  仍不可行, 则输出 “No Solution”, 否则输出  $S_{23}$  的最小值。时间复杂度为  $O(24^3 * \log_2 N)$ 。

这样题目就完全转化成了一个图论问题, 用经典的 Bellman Ford 算法判断是否存在负权回路。Bellman Ford 算法其实就是一种迭代算法, 图中的点不需要存在拓扑序列, 只要满足没有负权回路, 对于结点  $v_i$ , 用  $l_i$  表示当前从源点到该点的最短距离长度, 每次都更新它的子结点。由于一次更新至少能确定一个点, 所以在  $n-1$  次更新后  $n$  个点的值应该都确定了, 如果在第  $n$  次迭代后仍然有结点的  $l_i$  值被更新, 那么就可以得知图中含有负权回路, 否则  $l_i$  就是从源点到该点的最短路径长度。

下面给出 Bellman Ford 的算法描述:

Procedure Bellman Ford ( $G, S$ )

For  $I \leftarrow 1$  to  $n$  do  $l_i \leftarrow \infty$

$l_s \leftarrow 0$

Count  $\leftarrow 0$



```

repeat
  more  $\leftarrow$  false
  count  $\leftarrow$  count + 1
  for i  $\leftarrow$  1 to n do
    for j  $\leftarrow$  1 to n do
      if  $l_i + g_{i,j} < l_j$  then
         $l_j \leftarrow l_i + g_{i,j}$ 
        more  $\leftarrow$  true
  until not more or (count = n)
  if more then return (no solution) else return (L)

```

本题用到了差分约束系统的理论，巧妙地转化成图论的题目。像这类题目，元素之间有复杂的制约关系，拓扑关系不明确，很可能转化成有向无环图来解决。

### 例题 7-8 新桥。

问题描述：

一座城市有  $n$  ( $n \leq 5000$ ) 个小区，小区间有  $m$  ( $m \leq 100000$ ) 个双向行驶的道路，如图 7-19 所示。小区 A 的居民每天要到小区 B 或者小区 C 工作。为帮助大家节约路上的时间，市长打算新建一座长度不超过  $L_{\max}$  的新桥，使得新桥修建后：

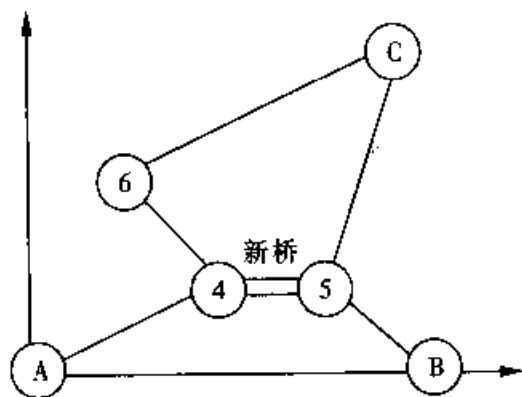


图 7-19 示例图

1. 小区 A 到小区 B 的最短路径比原有的路径短。
2. 小区 A 到小区 C 的最短路径比原有的路径短。
3. 在满足 1、2 的条件下使得两条最短路径的长度和最小。

输入：

第一行包括两个整数和一个实数： $N, M, L_{\max}$ ，表示有  $N$  个顶点， $M$  条边， $L_{\max}$  为长度限制。

接下来  $N$  行，第  $1+i$  行是一个整点坐标  $X_i, Y_i$ ，表示第  $i$  个顶点的坐标。其中 1 号点为 A，2 号和 3 号分别表示 B 和 C。

接下来  $M$  行，每行都有两个顶点编号，表示两个小区间有一条双向行驶道路。

输出：

两个顶点编号。表示新建的桥连接的两个小区。

样例:

Bridge. in	Bridge. out
6 5 40	4 5
0 0	
100 0	
100 100	
45 20	
55 20	
25 50	
1 4	
4 6	
5 2	
5 3	
6 3	

分析:

一个最容易想到的方法是:枚举新桥连接的两个小区  $u, v$ , 然后在加入边  $(u, v)$  后重新计算  $A-B$  和  $A-C$  的最短路长度。这样做的最坏情况时间复杂度为  $O(n^2m + n^3\log_2 n)$ , 不令人满意, 我们需要把算法加以改进。

可以注意到, 如果加入了新桥  $(u, v)$ , 那么新的  $A-B$ ,  $A-C$  必须要经过桥  $(u, v)$  才能满足条件。也就是说, 我们只需计算  $A-u, A-v, B-u, B-v$  的长度,  $A-B$  的最短路只可能是  $A-u-v-B$  或  $A-v-u-B$ 。这样做以后, 预处理如果用堆存储的 Dijkstra 复杂度是  $O(n + m\log_2 n)$ , 如果用二项堆的存储的 Dijkstra 复杂度是  $O(m + n\log_2 n)$ , 主过程的复杂度为  $O(n^2)$ , 总的复杂度为  $O(n^2 + m\log_2 n)$  或者  $O(n^2 + m)$ 。

还可以顺着这个思路进一步改进算法。假设新桥是  $(u, v)$ , 那么是否有可能  $A-B$  的最短路为  $A-u-v-B$  而  $A-C$  的最短路为  $A-v-u-C$  呢? 不可能! 因为新的两条路径长为  $S_1 = L(A, u) + L(u, v) + L(v, B) + L(A, v) + L(v, u) + L(u, C)$ , 而在没有桥之前的合法通路  $A-v-B$  和  $A-u-C$  的长  $S_2 = L(A, u) + L(u, C) + L(A, v) + L(v, B)$ 。而  $S_1 - S_2 = 2L(u, v) > 0$ , 因此两条新路长的和大于两条老路的长度和, 故这不可能是合法解。

这样, 任何一个合法解都可以表示为桥  $(u, v)$ , 而最优路径为  $A-u-v-B$  和  $A-u-v-C$ 。即枚举方式变为: 先枚举  $v$ , 求出  $v-B$  和  $v-C$ , 然后枚举  $u$ , 使得  $L(u, v) + L(A, u)$  尽量小。为了加速, 可以按照  $L(A, u)$  递增的顺序或者  $L(u, v)$  递减的顺序枚举。虽然最坏情况时间复杂度还是  $O(n^2 + m)$ , 但是效率有了一定的提高。

再来考虑一下  $L_{\max}$ 。如果可以建造桥的位置数目  $k$  不大, 那么如果不重复遗漏的枚举这  $k$  个位置, 复杂度将变为  $O(k + m + n\log_2 n)$ , 速度也将有很大提高。可以用分治法来解决这个问题。

从本题可以看出图论的基本算法应该灵活运用, 对于不同的题目具体分析, 可以根据具体情况适当的优化, 得到更加高效的算法。



## 7. 10 小 结

这一章中我们学习了多种用于查找的数据结构,介绍了图的概念和一些基本术语以及存储方法,并对图的遍历、最小生成树、最短路径、拓扑排序、关键路径等基本算法进行了详细的讲解。这些内容是解决图论问题必不可缺的,但是单单生搬硬套是不够的。因为建模后得到的问题是多种多样的,需要在对这些算法熟练掌握后,灵活运用,进行相应的改造和扩充,才能很好地解决问题。

### 习题七

#### 一、单选题

1. 设无向图的顶点个数为  $n$ , 则该图最多有( )条边。  
A.  $n-1$       B.  $n(n-1)/2$       C.  $n(n+1)/2$       D.  $n(n-1)$
2.  $N$  个顶点的连通图至少有( )条边。  
A.  $n-1$       B.  $n$       C.  $n+1$       D. 0
3. 在一个无向图中, 所有顶点的度数之和等于所有边数的( )倍。  
A. 3      B. 2      C. 1      D.  $1/2$
4.  $N$  个 ( $n > 1$ ) 顶点的强连通图中至少含有( )条有向边。  
A.  $n-1$       B.  $n$       C.  $n(n-1)/2$       D.  $n(n-1)$
5. 对于具有  $e$  条边的无向图, 它的邻接表中有( )个边结点。  
A.  $e-1$       B.  $e$       C.  $2(e-1)$       D.  $2e$
6. 具有  $n$  个顶点的有向无环图最多可包含( )条有向边。  
A.  $n-1$       B.  $n$       C.  $n(n-1)/2$       D.  $n(n-1)$
7. 一个有  $n$  个顶点和  $n$  条边的无向图一定是( )。  
A. 连通的      B. 不连通的      C. 无环的      D. 有环的
8. 为了实现图的广度优先搜索遍历, 其广度优先搜索算法需要使用的一个辅助数据结构为( )。  
A. 栈      B. 队列      C. 二叉树      D. 树
9. 为了保证一个有  $n$  ( $n \geq 3$ ) 个顶点的无向图是连通的, 这个图至少要有( )条边。  
A.  $(n-1) * (n-2)/2 + 1$       B.  $n * (n-1)/2 - (n+2)$   
C.  $(n-2) * (n-1)/2$       D.  $(n/2 + 1) * (n-1)$
10. 中序遍历为 ABC 的树有( )棵。  
A. 3      B. 1      C. 5      D. 8

#### 二、填空题

1. 在一个图中, 所有顶点的度数之和等于所有边数的\_\_\_\_\_倍。
2. 在一个具有  $n$  个顶点的无向完全图中, 包含有\_\_\_\_\_条边; 在一个具有  $n$  个顶点的

有向完全图中, 包含有\_\_\_\_\_条边。

3. 对于一个具有  $n$  个顶点的图, 若采用邻接矩阵表示, 则矩阵大小为\_\_\_\_\_。

4. 对于一个具有  $n$  个顶点和  $e$  条边的有向图和无向图, 在其对应的邻接表中, 所含边结点分别为\_\_\_\_\_和\_\_\_\_\_条。

5. 在有向图的邻接表和逆邻接表表示中, 每个顶点邻接表分别链接着该顶点的所有\_\_\_\_\_和\_\_\_\_\_结点。

6. 对于一个具有  $n$  个顶点和  $e$  条边的有向图和无向图, 若采用边集数组表示, 则存于数组中的边数分别为\_\_\_\_\_和\_\_\_\_\_条。

7. 对于一个具有  $n$  个顶点和  $e$  条边的无向图, 当分别采用邻接矩阵、邻接表和边集数组表示时, 求任一顶点度数的时间复杂度依次为\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_。

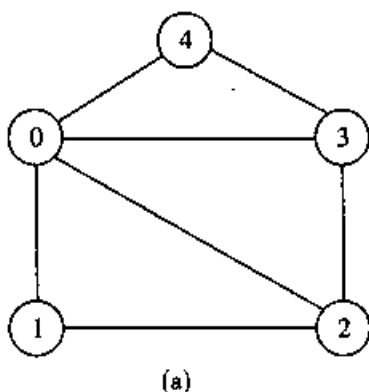
8. 假定一个图具有  $n$  个顶点和  $e$  条边, 则采用邻接矩阵、邻接表和边集数组表示时, 其相应的空间复杂度分别为\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_。

9. 对用邻接矩阵表示的图进行任一种遍历时, 其时间复杂度为\_\_\_\_\_; 对用邻接表表示的图进行任一种遍历时, 其时间复杂度为\_\_\_\_\_。

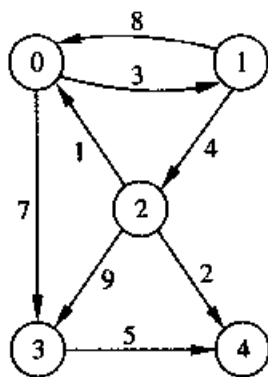
10. 对于一个具有  $n$  个顶点和  $e$  条边的连通图, 其生成树中的顶点数和边数分别为\_\_\_\_\_和\_\_\_\_\_。

### 三、运算题

1. 对于图 7-20 (a) 和 (b), 求出:



(a)



(b)

图 7-20 小题 1 及小题 2 的图

(1) 每一个图的二元组表示。对于带权图, 可将边的权写在该边的后面。

(2) (a) 图中每个顶点的度, 以及每个顶点的所有邻接点和所有边。

(3) (b) 图中每个顶点的入度、出度和度, 以及每个顶点的所有入边和出边。

(4) (a) 图中从  $V_0$  到  $V_4$  的所有简单路径及相应路径长度。

(5) (b) 图中从  $V_0$  到  $V_4$  的所有简单路径及相应带权路径长度。

2. 对于图 7-20 (a) 和 (b), 给出:

(1) 每个图的邻接矩阵。

(2) 每个图的邻接表。

(3) 每个图的边集数组。

3. 对于图 7-21 (a) 和 (b), 按下列条件分别写出从顶点  $v_0$  出发按深度优先搜索遍历得到

的顶点序列和按广度优先搜索遍历得到的顶点序列。

(1) 假定它们均采用邻接矩阵表示。

(2) 假定它们均采用邻接表表示, 并且假定每个顶点邻接表中的结点是按顶点序号从大到小的次序链接的。

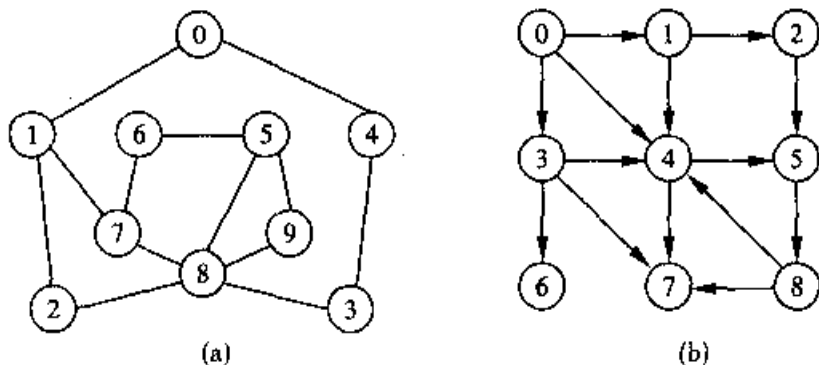


图 7-21 小题 3 的图

4. 已知一个带权图的顶点集  $V$  和边集  $E$  分别如下, 请利用克鲁斯卡尔算法求出该图的最小生成树的权, 以及依次得到的各条边。

$V = \{0, 1, 2, 3, 4, 5\};$

$E = \{(0, 1) 19, (0, 2) 21, (0, 3) 14, (1, 2) 16, (1, 5) 5, (2, 3) 26, (2, 4) 11, (3, 4) 18, (4, 5) 6\}.$

5. 假定一个 AOV 网的顶点集和边集为:

$V = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\};$

$E = \{ \langle 0, 2 \rangle, \langle 0, 3 \rangle, \langle 1, 3 \rangle, \langle 1, 4 \rangle, \langle 2, 3 \rangle, \langle 2, 5 \rangle, \langle 3, 5 \rangle, \langle 3, 6 \rangle, \langle 3, 8 \rangle, \langle 4, 6 \rangle, \langle 5, 7 \rangle, \langle 5, 8 \rangle, \langle 6, 8 \rangle, \langle 7, 9 \rangle, \langle 8, 9 \rangle \};$

试写出一种拓扑序列。若在它的邻接表存储结构中, 每个顶点邻接表中的边结点都是按照终点序号从大到小链接的, 则写出唯一一种拓扑序列。

#### 四、上机编程题

##### 1. 基础算法设计:

- 根据有向图的邻接矩阵  $GA$  求出序号为  $numb$  的顶点的度数。
- 根据无向图的邻接表  $GL$  求出序号为  $numb$  的顶点的度数。
- 求出一个用邻接矩阵  $GA$  表示的图中所有顶点的最大出度值。

##### 2. 海底之城。

问题描述:

C 国是一个高科技大国。目前, C 国投资巨额资金, 用于在深海处建设一座现代化高科技城市。当然, 这是一项长期而艰巨的工程。现在只是刚刚起步。

科技中心已经成功解决了初期建设的很多困难, 并且建立了一批海底基地。由于环境等复杂因素, 一个海底基地只能接收到其他某些海底基地发来的信息, 而如果基地 A 可以收到 B 发来的信息, 基地 C 可以收到 A 发来的信息, 那么基地 B 发出的信息就可通过 A 而被 C 收到。

现在, 控制中心准备在每个区域建立一个控制分站, 以更好地控制区域里的每个基地 (如果说基地 A、B 可以互通信息, 那么它们属于同一个区域)。这个分站要求设在每个区域内编号最小



的基地。

为了建立信息网络，控制中心需要知道每个基地所对应的控制分站属于哪个基地。他们知道解决这样的问题对于你来说是非常轻松的，所以把这个任务交给你单独完成。

输入：

文件第一行为一个数  $N, M$  ( $1 \leq N \leq 20000, 0 \leq M \leq 100000$ )，其中  $N$  表示海底基地的个数， $M$  表示接下的  $M$  行每行有一条描述。每条描述包括两个数  $A, B$  ( $1 \leq A, B \leq N$ )，表示基地  $A$  可以直接收到由基地  $B$  发来的信息。

输出：

一行，依次输出基地  $1 \sim N$  对应的控制分站所处基地的编号。

样例输入：

3 3

1 2

2 1

3 1

样例输出：

1 1 3

### 3. 繁忙的都市。

问题描述：

城市  $C$  是一个非常繁忙的大都市，城市中的道路十分拥挤，于是市长决定对其中的道路进行改造。城市  $C$  的道路是这样分布的：城市中有  $n$  个交叉路口，有些交叉路口之间有道路相连，两个交叉路口之间最多有一条道路相连接。这些道路是双向的，且把所有的交叉路口直接或间接地连接起来了。每条道路都有一个分值，分值越小表示这个道路越繁忙，越需要进行改造。但是市政府的资金有限，市长希望进行改造的道路越少越好，于是他提出下面的要求：

- (1) 改造的那些道路能够把所有的交叉路口直接或间接地连通起来。
- (2) 在满足要求 (1) 的情况下，改造的道路尽量少。
- (3) 在满足要求 (1)、(2) 的情况下，改造的那些道路中分值最大的道路分值尽量小。

任务：

作为市规划局的你，应当作出最佳的决策，选择哪些道路进行修建。

输入：

文件名：city.in，

第一行有两个整数  $n, m$  表示城市有  $n$  个交叉路口， $m$  条道路。接下来  $m$  行是对每条道路的描述， $u, v, c$  表示交叉路口  $u$  和  $v$  之间有道路相连，分值为  $c$ 。( $1 \leq n \leq 300, 1 \leq c \leq 10000$ )

输出：

文件名：city.out，

两个整数  $s, \max$ ，表示你选出了几条道路，分值最大的那条道路的分值是多少。

数据范围：

$1 \leq n \leq 300, 1 \leq c \leq 10000$

样例：

city.in

city. out

4 5

1 2 3

1 4 5

2 4 7

2 3 6

3 4 8

3 6

#### 4. 小朋友分组问题:

幼儿园有  $n$  个小朋友, 他们互相之间或许认识, 或许不认识, 现在给定  $n$  及小朋友们的交友关系, 要求把所有的小朋友划分成两组, 使得每一组之间的人互相之间都认识。求一种方案, 要求两组人数相差最小。

5. 设有  $n$  个城市, 依次编号为  $1, 2, \dots, n$ , 另外有一个文件保存着  $n$  个城市之间的距离。当两个城市之间距离等于  $-1$  时, 表示这两个城市没有直接连接。求指定城市  $k$  到每个城市  $i$  ( $1 \leq k, i \leq n \leq 50$ ) 的最短距离和对应的最短路径。

输入:

输入文件: 第一行保存两个整数, 第一个是城市数目, 第二个是指定城市的编号。

以下  $n$  行为一个  $n \times n$  的邻接矩阵。

输出:

输出文件有  $n$  行, 第  $i$  行代表指定城市  $k$  到  $i$  最短距离和最短路径, 中间用空格隔开,  $k$  到自己的距离为  $0$ 。

样例:

输入:

5 3

0 40 65 -1 30

40 0 105 -1 -1

65 105 0 10 35

-1 -1 10 0 20

30 -1 35 20 0

输出:

60 3 ->4 ->5 ->1

100 3 ->4 ->5 ->1 ->2

0

10 3 ->4

30 3 ->4 ->5

## 8 查找

### 8.1 查找的基本概念

查找 (search) 又称检索。它同人们的日常工作和生活有着密切地联系。如从图书馆查找书籍, 从成绩表中查找成绩排名, 从电话簿查找电话号码, 在搜索网站中搜索资料等。当今社会是一个信息爆炸的社会, 在庞大的信息量中寻找实用的信息是一项必需的技能, 而计算机是一种查找信息的利器。

利用计算机查找需要把人工组织的信息表 (或称数据表) 存入计算机中, 变为计算机可利用的“表”, 查找过程就在这个表上进行。如利用计算机进行图书管理, 就把人工整理的图书目录表存入计算机中, 变为计算机可利用的表, 查找时根据用户提供的书名或其他信息, 从这个表中查找出所需要的图书。在计算机中, 作为查找对象的表是指数据的存储结构, 不同存储结构对应着不同的表。如线性表 (即为带有线性结构的数据) 的顺序存储结构对应着顺序表, 其链接存储结构对应着单链表 (假定不考虑双向链表), 二叉排序树 (即为带有二叉树结构的数据) 的链接存储结构对应着二叉链表等。这一章还将结合查找运算介绍线性表的索引存储结构 (对应索引表和主表) 和散列存储结构 (对应散列表) 等内容。

在计算机上对表进行查找, 就是根据所给条件查找出满足条件的第一条记录 (元素) 或全部记录, 若没有找到满足条件的任何记录, 则返回特定值, 表明查找失败; 若查找到满足条件的第一条记录, 则表明查找成功, 返回该记录的存储位置, 以便对该记录作进一步处理; 若需要查找到满足条件的所有记录, 则可看作为在多个区间内连续查找到满足条件的第一条记录的过程, 即首先在整个区间内查找到满足条件的第一条记录, 接着在剩余的区间内查找到满足条件的第一条记录 (对整个区间而言, 它是满足条件的第二条记录), 依此类推, 直到剩余区间为空时止。所以, 查找问题就归结为在指定的区间 (即表的一部分或全部) 内查找满足所给条件的第一条记录, 若查找成功, 则返回记录的存储位置, 否则表明查找失败, 返回一个特定值。当然, 查找运算只是整个数据处理过程的一个环节, 它的下一个环节是如何对查找结果进行处理, 这可根据实际需要, 对查找成功的记录进行观察、计算、输出、修改、删除等, 当查找失败时, 输出错误信息或插入新记录等。

用于在表上查找记录的条件, 情况比较复杂, 它由具体应用而定, 但其中最具有代表性的条件是: 在关键字段 (项) 上查找关键字等于给定值  $K$  所在的记录。由于表中每个记录的关键字都不同, 所以这种条件只可能查找到唯一的一条记录。在本章的讨论中, 我们将以这种条件为依据给出各种查找的算法, 当然读者也不难根据实际需要给出使用其他条件的查找算法。

作为查找对象的表的结构不同, 其查找方法一般也不同。但无论哪一种方法, 其查找过程都是用给定值  $K$  同关键项上的关键字按照一定的次序进行比较的过程, 比较次数的多少就是相应算法的时间复杂性, 它是衡量一个查找算法优劣的重要指标。



## 8.2 顺序表查找

顺序表 (sequential list) 是指线性表的顺序存储结构。顺序存储线性表结构的类型定义已经在前面的章节给出, 它包含两个域: 顺序存储线性表元素的向量域和存储线性表长度的指针域。

在程序实现过程中, 顺序存储线性表结构一般用数组实现。例如: 用一个一维数组来实现顺序存储, 那么存储形式如下:

	1	2	...	n	...	m
A	$a_1$	$a_2$	...	$a_n$	...	

表中  $m$  表示数组空间的上届,  $n$  表示线性实际的长度。顺序存储线性表元素的数组和数组长度  $n$  可以这样说明:

A : array [1..m] of elemtype;

n : integer;

其中 elemtype 可以为任何类型, 假设是下面的抽象记录类型:

elemtype = record

key : keytype;

...

end ;

对于一种元素类型可以有多个关键字域 (key), 关键字可以是任意类型。在进行查找的过程中, 关键字就是查找对象的基准, 用关键字来判断是否查找到所需的对象。

由于这种说明方式在表述上十分清晰, 所以在后面的文章的举例和伪代码中, 将都会采用这种说明方式表示一个顺序存储线性表。

在顺序表上进行查找有多种方法, 这里只介绍最主要的两种方法——顺序查找和二分查找。

### 一、顺序查找

顺序查找 (sequential search) 是一种最简单和最基本的查找方法。它的基本思想是: 从顺序表的一端开始, 依次将每个元素的关键字同目标值  $K$  进行比较, 若某个元素的关键字等于目标值  $K$ , 则表明查找成功, 返回该元素所在域的指针 (在数组中就是数组下表); 若所有的表中元素比较了一遍都没有找到目标值  $K$ , 那么说明表中并没有这个目标值的元素。

顺序查找的伪代码:

Procedure Sequential Search (A, n, k)

$i \leftarrow 1$

  while ( $i \leq n$ ) and ( $A[i] \neq K$ ) do

$i \leftarrow i + 1$

  if  $i \leq n$

```
then return (i)
else return (0)
```

在上面的代码中每次循环都要进行两个判断。为了省去前一个判断，可以在表的另一端事先设置“哨岗”，即把给定值  $K$  赋给第  $n+1$  单元（对应从表头开始向后查找）或第 0 单元（对应从表尾开始向前查找）的关键字域，当查找失败即查找到第  $n+1$  单元或第 0 单元时，因比较相等而使循环正常结束。改进后的算法描述如下：

Procedure Sequential Search ( $A, n, k$ )

```
A[n+1].key ← K
i ← 1
while A[i] ≠ K do
    i ← i + 1
if i ≤ n
    then return (i)
    else return (0)
```

顺序查找的时间复杂度为  $O(n)$ ，速度较慢，效率不高，但是实现方式是最为简单的。顺序查找适用于顺序线性表，同样适用于单链表，而且对表中元素的排列次序无要求。这将给插入新元素带来方便，因为不需要为新元素寻找插入位置和移动原有元素，只要把它加入到表尾（对于顺序表）或表头（对于单链表）即可。

为了提高顺序查找的速度，一种可考虑的方法是，在已知各元素查找概率不等的情况下，可将各元素按查找概率从大到小排列，从而降低查找的平均比较次数；另一种可考虑的方法是，在事先未知各元素查找概率的情况下，在每次查找到一个元素时，将它与前驱元素对调位置，这样，查找频度高（即概率大）的元素就会逐渐前移，最后形成元素的前后位置按查找概率从大到小排列，从而达到减少平均比较次数的目的。

## 二、二分查找

二分查找 (binary search) 又称折半查找。作为二分查找对象的表必须是顺序存储的有序表，通常假定有序表是按关键字从小到大有序（或者任何一种有序方式，只要这种有序方式是单调的）。二分查找的过程是：首先取整个有序表  $A(1..n)$  的中点元素  $A[mid]$ （其中  $mid = \lfloor \frac{n+1}{2} \rfloor$ ）的关键字同给定值  $K$  比较，若相等，则查找成功，返回该元素的下标  $mid$ ；否则，若  $A[mid].key > K$ ，则说明待查元素（即关键字等于  $K$  的元素）只可能落在左子表  $A(1..mid-1)$  中，接着只要在左子表中继续进行二分查找即可，若  $A[mid].key < K$ ，则说明待查元素只可能落在右子表  $A(mid+1..n)$  中，接着只要在右子表中继续进行二分查找即可；这样，经过一次关键字的比较，就缩小一半查找空间，如此进行下去，直到找到关键字为  $K$  的元素，或者当前查找区间为空（说明表中不存在关键字为  $K$  的元素）时止。

二分查找的过程是递归的，但是很容易用非递归算法描述。（递归算法留给读者自行思考。）

Procedure Binary Search ( $A, n, K$ )

```
low ← 1
high ← n
```

```

while low ≤ high do
    mid ← (low + high) / 2
    case
        A[mid].key = K; return (mid)
        A[mid].key > K : high ← mid - 1
        A[mid].key < K : low ← mid + 1
    end case
return (0)

```

**例题 8-1** 假定有序表 A 中 8 个元素 (即  $n=8$ ) 的关键字序列为:

13, 23, 26, 47, 89, 101, 111, 123

现在要在这个有序表中查找 26 这个元素, 那么查找过程如下:

括号表示当前查找的区间, “[” 右边第一个元素为 low, “)” 左边第一个元素为 high, 箭头表示 mid 所在的位置。

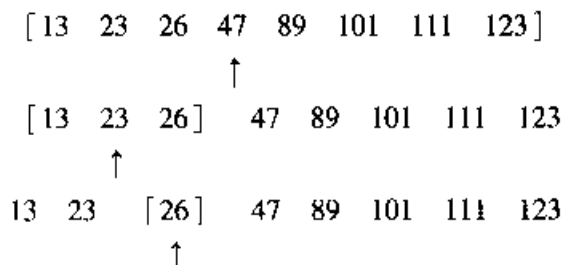


图 8-1 二分查找的过程

三次比较后成功地找到了 26 这个元素。

二分查找过程可用一棵二叉树来描述, 树中的每个根结点对应当前查找区间的中点元素  $A[mid]$ , 它的左子树和右子树分别对应该区间的左子表和右子表, 通常把此二叉树称为二分查找的判定树。由于二分查找是在有序表上进行的, 所以其对应的判定树必然是一棵二叉排序树。图 8-2 就是一棵描述图 8-1 查找过程的判定树, 树中每个结点的值为对应元素的关键字, 结点上面的数字为对应元素的下标, 附加的带箭头虚线表示查找一个元素的路径, 其中给出了图 8-1 中查找关键字为 26 元素的路径。从此图可以清楚地看出, 在有序表上二分查找一个关键字等于  $K$  的元素时, 对应着判定树中从树根结点到待查结点的一条路径, 同关键字进行比较的次数就等于该路径上的结点数, 或者说等于待查结点的层数。

进行二分查找的判定树不仅是一棵二叉排序树, 而且是一棵理想平衡树, 因为它除最后一层外, 其余所有层的结点数都是满的, 所以判定树的高度  $h$  和结点数  $n$  之间的关系为:  $h = \lfloor \log_2 n \rfloor + 1$ 。

这就告诉我们, 二分查找成功时, 同元素关键字进行比较的次数最多为  $h$ , 在等概率的情况下平均比较次数略低于  $h$  (证明从略), 所以二分查找算法的时间复杂性为  $O(\log_2 n)$ 。显然它比顺序查找的速度要快得多。例如, 假定一个有序表含有 1000 个元素, 若采用二分查找则至多比较 10 次; 若采用顺序查找, 则最多需要比较 1000 次, 平均也得比较 500 多次。

在二分查找中, 查找失败也对应着判定树中的一条路径, 它是从树根结点到相应结点的空子树。当待查区间为空, 即区间上界小于区间下界时, 比较过程就达到了这个空子树。由此可知, 二分查找失败时, 同关键字进行比较的次数也不会超过树的高度, 所以不管二分查找成功与失败, 其时间复杂性均为  $O(\log_2 n)$ 。

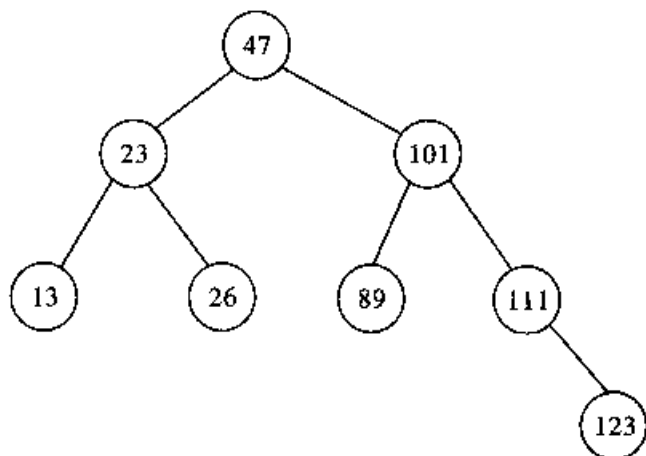


图 8-2 二分查找的判定树

二分查找的优点是比较次数少, 查找速度快。但在查找之前要为建立有序表付出代价, 同时对有序表的插入和删除都需要平均比较和移动表中的一半元素, 是很浪费时间的操作。所以, 二分查找适用于数据相对固定的情况。

**例题 8-2 集合 (经典例题)。**

给定两个集合 A、B, 集合内的任一元素  $x$  满足  $1 \leq x \leq 10^9$ , 并且每个集合的元素个数不大于  $10^5$ 。我们希望求出 A、B 之间的关系。(内存: 2M)

根据给定两个集合的描述, 判断它们满足下列关系的哪一种:

A 是 B 的一个真子集, 输出 "A is a proper subset of B"

B 是 A 的一个真子集, 输出 "B is a proper subset of A"

A 和 B 是同一个集合, 输出 "A equals B"

A 和 B 的交集为空, 输出 "A and B are disjoint"

上述情况都不是, 输出 "I'm confused!"

分析:

要判断集合 A 和 B 的关系, 只需要看 B 中的每个元素和集合 A 的关系, 归结起来下面是判断 5 种情况的充要条件:

1. A 是 B 的一个真子集: ——判断 B 中的元素是否在 A 中, 如果在 A 中, 那么就将 A 中相应的元素进行标记。若最后 A 中所有元素都被标记, 且 B 中至少有一个不在 A 中, 则 A 是 B 的真子集。

2. B 是 A 的一个真子集: ——判断 B 中的元素是否在 A 中, 如果在 A 中, 那么就将 A 中相应的元素进行标记。若 B 中所有元素都在 A 中, 且 A 中所有元素没有被标记, 则 B 是 A 的一个真子集。

3. A 和 B 是同一个集合: ——判断 B 中的元素是否在 A 中, 如果在 A 中, 那么就将 A 中相应的元素进行标记。若 B 中所有元素都在 A 中, 且 A 中所有元素都被标记, 则 B 和 A 是同一个集合。

4. A 和 B 的交集为空: B 中没有元素在 A 中。

5. 上述情况都不是, 就输出 "I'm confused!"

可以看出只要对 B 的元素在 A 中进行查找即可。因此对 A 中的元素进行从小到大排序, 删除重复的元素。对 A 初始化后, A 就有序化了, 那么可以利用二分查找判断 B 中的每个元素与集合 A 的关系。

排序的时间复杂度为  $O(n \log_2 n)$ , 二分查找的复杂度为  $O(n \log_2 n)$ , 所以总的复杂度为  $O(n \log_2 n)$ 。

## 8.3 索引查找

### 一、索引查找

索引查找 (index search) 又称分级查找。例如, 在汉语字典中查找汉字, 如果知道字形, 现在部首表中找到对应检字表中对应的页码, 然后再根据字的笔画顺序查找到对应正文中的页码, 然后在此页码中便找到了需要查找的汉字。在这里, 整个字典就是索引查找的对象, 字典的正文是字典的主要部分, 被称之为主表, 检字表、部首表和音节表都是为了方便查找主表而建立的索引, 所以称之为索引表。

同样在计算机中, 索引查找就是建立在索引存储结构的基础上进行的。索引存储的基本思想是: 首先把一个线性表 (主表) 中的元素按照一定的函数关系或条件划分成若干个子表, 每个子表建立一个索引项, 所有这些索引项构成主表的一个索引表。然后可以用顺序或链接的方式来存储索引表和每个子表。索引表中的每个元素一般包含三个域: 索引值域、指向子表的地址、指向子表的长度。相应的类型定义描述如下:

```
Indexitem = record
```

```
Index ; 索引值的类型 (可以是任意的函数值: 字符、数字)
```

```
Address ; integer; (这是假设子表是静态顺序结构存储, 若为动态链接, 那么这里一个为指针)
```

```
Length ; integer;
```

```
end ;
```

```
indexlist = array [1..m] of indexitem;
```

例如, 要为参加雅典奥运会的运动员资料建立一个索引表。可以用运动员的年龄、参赛项目、编号等信息建立索引表。假设用运动员的项目编号进行分类, 运动员资料如下:

编 号	参赛项目	运动员
TF001	田径	刘翔
ST004	射击	杜丽
ST001	射击	朱启南
DV001	跳水	田亮
DV002	跳水	彭勃
DV003	跳水	胡佳
SW006	游泳	罗雪娟
BB001	篮球	姚明
BB002	篮球	李楠
BB003	篮球	刘炜

那么根据编号建立一个线性表, 这个线性表作为主表, 如下:

Mainlist = (TF001, ST004, ST001, DV001, DV002, DV003, SW006, BB01, BB002, BB003)

然后根据项目值进行分类:

TF = (TF001)

ST = (ST001, ST004)

DV = (DV001, DV002, DV003)

SW = (SW006)

BB = (BB001, BB002, BB003)

这样根据项目建立起一个索引表:

Indexlist	Index	Address	Length
1	TF	1	1
2	ST	2	2
3	DV	4	3
4	SW	7	1
5	BB	8	3

其中 Address 域表示每个项目运动员在主表中出现的第一个位置。Length 表示运动员的个数,也就是这个项目的运动在主表中占据的长度。

建立动态链接的示意图如下:

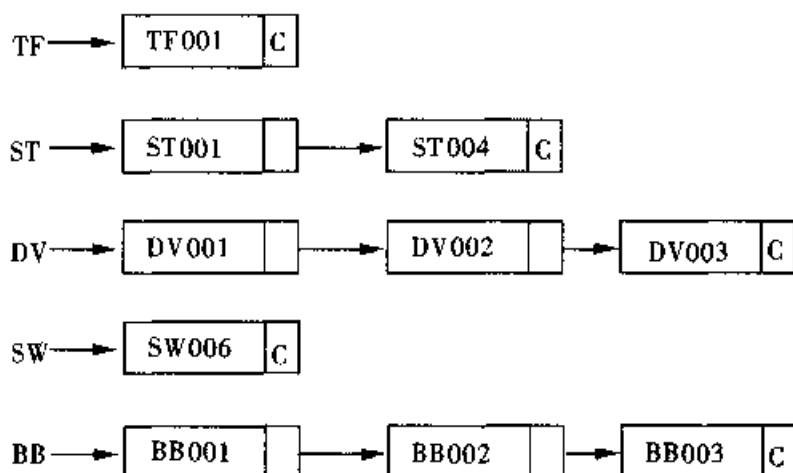


图 8-3 索引查找的动态链接

## 二、索引查找算法

索引查找是在索引表(即线性表的索引存储结构)上进行的查找。索引查找的过程是:首先根据给定的索引值  $K_1$ , 在索引表上找到索引项为  $K_1$  的索引项, 以确定其子表所在的位置; 然后再根据关键字  $K_2$ , 在对应的字表中查找关键字等于  $K_2$  的元素。在索引表和子表中查找的过程中既可以用顺序查找, 又可以用二分查找。

设数组  $A$  是主表,  $B$  是索引表。  $m$  表示索引表的长度。算法描述如下:

Procedure Index Search (A, B, m, K1, K2)

$i \leftarrow \text{Sequential Search}(B, m, K1)$  (在索引表中进行顺序查找)

if  $i = 0$  then return (0) (说明查找失败)

$j \leftarrow B[i].\text{address}$

while ( $j < B[i].\text{address} + B[i].\text{length}$ ) and ( $K2 \neq A[j].\text{key}$ ) do

$j \leftarrow j + 1$

if  $j = B[i].\text{address} + B[i].\text{length}$

then return (0)

else return (j)

索引查找的比较次数等于算法中查找索引表的比较次数和查找相应子表的比较次数之和。查找索引表最多比较  $m$  次, 查找相应的子表 (为了便于讨论, 假定每个子表具有相同的长度, 其长度为  $n/m$ ,  $n$  为主表中的元素个数) 的最多比较次数为  $n/m$ , 所以整个索引查找过程的最多比较次数为  $m + n/m$ 。若对  $m$  求导得  $1 - \frac{n}{m^2}$ , 令导函数等于 0, 可得当  $m$  取  $\sqrt{n}$  时, 原函数  $m + n/m$  取最小值  $2\sqrt{n}$ , 也就是说, 当把线性表中的  $n$  个元素划分为  $\sqrt{n}$  个等长的子表 (每个子表的长度也为  $\sqrt{n}$ ) 时, 索引查找的最多比较次数的值最小, 为  $2\sqrt{n}$ 。读者容易证明, 在查找各元素概率相等的情况下, 此时的平均比较次数也为最少, 即为  $\sqrt{n}$ 。所以索引查找算法 (假定索引表和子表都采用顺序查找) 的时间复杂性为  $O(m + n/m)$ , 特别地, 当  $m = \sqrt{n}$  时, 其时间复杂性为  $O(\sqrt{n})$ 。

## 8.4 散列查找

### 一、散列的概念

散列 (hash) 同顺序、链接和索引一样, 是存储线性表的又一种方式。散列存储的基本思想是: 以线性表中的每个元素的关键字  $K$  为自变量, 通过一种函数  $h(K)$  计算出函数值, 把这个值解释为一块连续存储空间 (即数组空间) 的单元地址 (即下标), 将该元素存储到这个单元中。散列存储中使用的函数  $h(K)$ , 称为散列函数或哈希函数, 它实现关键字到存储地址的映射 (或称转换),  $h(K)$  的值称为散列地址或哈希地址,  $f$  使用的数组空间是线性表进行散列存储的地址空间, 所以被称之为散列表 (hashlist) 或哈希表。在散列表上进行查找时, 首先根据给定的关键字  $K$ , 用与散列存储时使用的同一散列函数  $h(K)$  计算出散列地址, 然后按此地址从散列表中取出对应的元素。

**例题 8-3** 假定一个线性表为:

$A = (18, 75, 60, 43, 54, 90, 46)$

为了散列存储该线性表, 假定选取的散列函数为:

$h(K) = K \bmod m$

即用元素的关键字  $K$  除以散列表的长度  $m$ , 取余数 (即为 0 至  $m-1$  范围内的一个数) 作为存

储该元素的散列地址,这里假定  $K$  和  $m$  均为正整数,并且  $m$  要大于等于待散列的线性表的长度  $n$ 。在此例中,  $n=7$ ,所以假定取  $m=13$ ,则得到的每个元素的散列地址为:

$$\begin{aligned} h(18) &= 18 \bmod 13 = 5 & h(75) &= 75 \bmod 13 = 10 \\ h(60) &= 60 \bmod 13 = 8 & h(43) &= 43 \bmod 13 = 4 \\ h(54) &= 54 \bmod 13 = 2 & h(90) &= 90 \bmod 13 = 12 \\ h(46) &= 46 \bmod 13 = 7 \end{aligned}$$

若根据散列地址把元素存储到散列表  $H(0:m-1)$  中,则存储映象为:

	0	1	2	3	4	5	6	7	8	9	10	11	12
H				54		43	18		46	60		75	90

从散列表中查找元素同插入元素一样简单,如从  $H$  中查找关键字为 60 的元素时,只要利用上面的函数  $h(K)$  计算出  $K=60$  时的散列地址 8,则从下标为 8 的单元中取出该元素即可。

上面讨论的散列表是一种理想的情况,即插入时根据元素的关键字求出的散列地址,其对应的存储单元都是空闲的,也就是说,每个元素都能够直接存储到它的散列地址所对应的单元中,不会出现该单元已被其他元素占用的情况。在实际应用中,这种理想情况是很少见的,通常可能出现一个待插入元素的散列地址单元已被占用,使得该元素无法直接存入到此单元中,我们把这种现象叫做冲突 (collision)。在散列存储中,冲突是很难避免的,除非关键字的变化区间小于等于散列地址的变化区间,而这种情况当关键字取值不连续时又是非常浪费存储空间的,一般情况是关键字的取值区间大于散列地址的变化区间 (即散列函数的数值范围)。如在例题 8-3 中,关键字为两位正整数,其取值区间为  $0 \sim 99$ ,而散列地址的取值区间为  $0 \sim 12$ ,远比关键字的取值区间小。这样,当不同的关键字通过同一散列函数计算散列地址时,就可能出现具有相同散列地址的情况,若该地址中已经存入了一个元素,则具有相同散列地址的其他元素就无法直接存入进去,从而引起冲突。通常把这种具有不同关键字而具有相同散列地址的元素称做“同义词”,由同义词引起的冲突称做同义词冲突。如再向例题 8-3 的散列表  $H$  中插入一个关键字为 70 的新元素时,该元素的散列地址为 5,就同已存入的关键字为 18 的元素发生冲突,致使关键字为 70 的新元素无法存入到下标为 5 的单元中。因此,如何尽量避免冲突和冲突发生后如何解决冲突 (即为发生冲突的待插入元素找到一个空闲单元、使之存储起来) 就成了散列存储的两个关键问题。

在散列存储中,虽然冲突很难避免,但发生冲突的可能性却有大有小,这主要与三个因素有关。一是与装填因子  $\alpha$  有关,所谓装填因子是指散列表中已存入的元素数  $n$  与散列表空间大小  $m$  的比值,即  $\alpha = n/m$ ,当  $\alpha$  越小时,冲突的可能性就越小, $\alpha$  越大 (最大取 1) 时,冲突的可能性就越大。这很容易理解,因为  $\alpha$  越小,散列表中空闲单元的比例就越大,所以待插入元素同已存元素发生冲突的可能性就越小;反之, $\alpha$  越大,散列表中空闲单元的比例就越小,所以待插入元素同已存元素冲突的可能性就越大。另一方面, $\alpha$  越小,存储空间的利用率也就越低;反之,存储空间的利用率也就越高。为了既兼顾减少冲突的发生,又兼顾提高存储空间的利用率这两个方面,通常使  $\alpha$  最终控制在  $0.6 \sim 0.9$  范围内为宜。二是与所采用的散列函数有关,若散列函数选择得当,就能够使散列地址尽可能均匀地分布在散列空间上,从而减少冲突的发生;否则,若散列函数选择不当,就可能使散列地址集中于某些区域,从而加大冲突的发生。三是与解决冲突的方法有关,方法选择的好坏也将减少或增加发生冲突的可能性。后面将陆续讨论影响冲突发生的这三个因素。



## 二、散列函数

构造散列函数的目标是使散列地址尽可能均匀地分布在散列空间上,同时使计算尽可能简单,以节省计算时间。根据关键字的结构和分布不同,可构造出与之适应的各不相同的散列函数,这里只介绍较常用的几种,其中又以介绍除留余数法为主。在下面的讨论中,假定关键字均为整型,若关键字不为整型,则要设法把它转换为整型。

### 1. 直接定址法

直接定址法是以关键字  $K$  本身或关键字加上某个数值常量  $C$  作为散列地址的方法。对应的散列函数  $h(K)$  为:

$$h(K) = K + C$$

这种方法计算最简单,并且没有冲突发生。它适用于关键字的分布基本连续且关键字范围比较小的情况。否则容易造成空间上的浪费。

### 2. 除留余数法

除留余数法在信息学竞赛中是最常使用的一种方法。就是上一节例子中举出的散列函数,将关键字除一个数字取余。函数式为:

$$h(K) = K \bmod m$$

这种方法的关键是选好  $m$ ,使得每一个关键字通过该函数转换后映射到散列空间上任一地址的概率都相等,从而减少冲突的可能性。效果最好的是当  $m$  为一个质数。

### 3. 数字分析法

数字分析法是取关键字中某些取值较分散的数字位作为散列地址的方法。这个方法适合于关键字比较大,比如是一个高精度数或者一个字符串,因为没有必要也不可能把整个数串作为哈希值,只需要将其中某些数字位作为散列地址就可以了。比如为(9231602, 92326875, 92739628)设计哈希函数,可以发现数字串的第4,6,7位分布比较散,那么用这几位作为哈希函数值,冲突就比较少。

## 三、处理冲突的方法

### 1. 线性开型寻址法

所谓开型寻址法就是从发生冲突的那个单元开始,按照一定的顺序,从散列表中查找出一个空闲的存储单元,把发生冲突的待插入元素存入到该单元中的一类处理冲突的方法。其中线性开型寻址法是开型寻址法中一种最简单的探查方法。它从发生冲突的  $d$  单元起,依次探查下一个元素(可以把表看成一个环,当达到表尾单元的时候,下一个探查的单元是表首单元),直到碰到一个空闲单元为止。

例如,在图8-4中给出一个散列表  $ht$ ,桶号从0到10。表中有3个元素,除数  $D$  为11。因为  $80 = 3 \pmod{11}$ ,则80的位置为3,  $40 = 7 \pmod{11}$ ,  $65 = 10 \pmod{11}$ 。每个元素都在相应的桶中。散列表中余下的单元为空。然后要将一个新的元素58加入散列表中。58的位置是  $3 \pmod{11}$ ,这个单元已经被占据了,那么就查找下一个单元  $3+1=4$ ,发现这个单元是空的,那么把58放在单元4中。现在要插入98,其起始桶10已满,则插入下一个可用桶0中。由此看来,在寻找下一个可用桶时,表被视为环形的。

			80				40			65	
Ht	0	1	2	3	4	5	6	7	8	9	10

			80	58			40			65	
Ht	0	1	2	3	4	5	6	7	8	9	10

98			80	58			40			65	
Ht	0	1	2	3	4	5	6	7	8	9	10

图 8-4 开型寻址法的插入示例

利用线性探查法处理冲突容易造成元素的“堆积”（或称“聚集”），这是因为当连续  $n$  个单元被占用时，再散列到这些单元上的元素和直接散列到后面一个空闲单元上的元素都要占用这个空闲单元，致使该空闲单元很容易被占用，造成堆积，并且堆积现象会越来越严重，从而大大地增加查找下一个空闲单元的路径长度。

在线性探查中，造成堆积现象的根本原因是探查序列过分集中在发生冲突的单元的后面，没有在整个散列空间上分散开。为了克服这个缺点，自然想到使用步长为 2, 3 等的线性探查法，取代上面讨论的步长为 1 的线性探查法，使得探查序列尽量分散。如步长取 2 时，探查序列为  $d, d+2, d+4, \dots$ ，或写成  $(d+2i) \bmod m$  ( $0 \leq i \leq m-1$ )，此探查序列的地址分布比步长取 1 时较分散。造成堆积的程度比步长取 1 时要有所减轻；步长取 3 时，探查序列为  $(d+3i) \bmod m$  ( $0 \leq i \leq m-1$ )，此探查序列的地址分布又比步长取 2 时较分散，造成堆积的程度将比步长取 2 时又有所减轻。当然不是说，步长取得越大越好，若散列表的长度为  $m$ ，则通常取靠近的整数值作为步长时效果最好，它接近于探查序列在整个散列空间上随机分布的情况。进行线性探查取步长值  $s$  时，必须避免  $m$  被  $s$  整除的情况出现。因为若出现这种情况，探查序列只在下标为  $(d+is) \bmod m$  的  $t$  个单元 ( $0 \leq i < t$ ，其中  $t$  是  $m$  被  $s$  整除所得的商) 上循环，不能探查到散列表中的其他单元。

## 2. 链表散列法

当散列发生溢出时，链表是一种好的解决方法。图 8-5 给出了散列表在发生溢出时采用链表来进行解决的方法。在上一个例子中，散列函数的除数为 11。在该散列表的组织中，每个桶仅含有一个结点指针，所有的元素都存储在该指针所指向的链表中。在搜索关键字值为  $K$  的元素时，首先要计算其起始桶。起始桶号为  $K \bmod d$ ，然后搜索该桶所对应的链表。在插入时，首先要保证表中不含有相同关键字的元素。当然，此时的搜索仅限于该元素的起始桶所对应的链表。由于每次插入都要首先进行一次搜索，因此把链表按照升序排列比无序排列会更有效。最后，为了删除关键字值为  $K$  的元素，首先访问起始桶对应的链表，找到该元素，然后删除。

## 四、散列表的插入和插入算法

在线性表的散列存储中，处理冲突的方法不同，其散列表的类型定义也不同。若采用开型寻址法则类型定义为：

hashlist1 = array [0..m-1] of elemtype;

若采用动态链接法，则类型定义为：

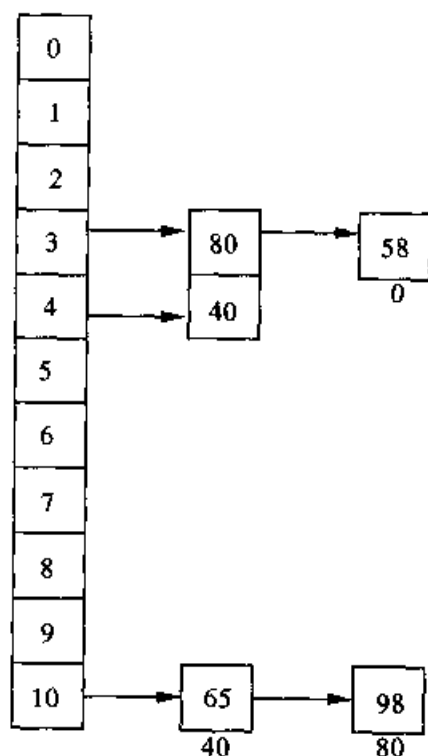


图 8-5 链表型的哈希表

hashlist2 = array [0..m-1] of dynanode;

其中 dynanode 是单链表类型, 可以为动态链表, 也可以为静态链表。

在上面每一种类型的散列表中, 散列表的长度  $m$  都要大于等于待散列的线性表的长度  $n$ , 当然最好取  $n$  与  $m$  的比值 (即装填因子  $a$  的值) 在 0.6~0.9 之间。

下面分别以 hashlist1 和 hashlist2 类型的散列表为例, 给出相应的插入和删除算法。

#### 1. 具有 hashlist1 类型的散列表的插入算法

Procedure Insert1 (HA, m, x);

$d \leftarrow h(x.key)$  ( $h$  为哈希函数)

while HA[d].key  $\neq$  null do

$d \leftarrow (d + a) \bmod m$  ( $a$  为插入时处理冲突的步长)

HA[d]  $\leftarrow$  x

#### 2. 具有 hashlist1 类型的散列表的查找算法

Procedure Search1 (Ha, m, K)

$d \leftarrow h(K)$

while (HA[d].key  $\neq$  null) and (HA[d].key  $\neq$  K) do

$d \leftarrow (d + a) \bmod m$

if HA[d].key = null

then return (m) (如果找到空闲单元, 说明散列表中不存在待查元素)

else return (d)

#### 3. 具有 hashlist2 类型的散列表的插入算法

Procedure Insert2 (HB, m, x)

```

d ← h (x. key)
if HB [d] = null
    then
        HB [d] ^ data ← x
        HB [d] ^ next ← null
    else
        new (p)
        p ^ data ← x
        p ^ next ← HB [d] . next
        HB [d] . next ← p
    
```

#### 4. 具有 hashlist2 类型的散列表的查找算法

Procedure Search2 (HB, m, K)

```

d ← h (K)
If HB [d] = null
    then return (m) (找不到为 K 的元素)
p ← HB [d]
while (p ^ data. key ≠ K) and (p ^ next ≠ null) do
    p ← p ^ next
if p ^ data. key = K
    then return (p) (若查找成功, 那么返回待查元素所在的结点)
    else return (null)
    
```

在散列存储中, 插入和查找的速度是相当快的, 它优于前面介绍过的任何一种方法, 特别是当数据量很大时更是如此。散列表插入、删除和查找的平均复杂度为  $O(1)$ 。散列存储的缺点是: ①占用的存储空间较多, 因为总是取  $a$  值小于 1; ②若采用开型定址法处理冲突, 则给删除元素带来困难, 若把被删除元素所占用的单元置空, 则割断了元素的查找路径, 所以只能给被删除的元素附加删除标记, 这又造成该单元的浪费; ③只能按关键字查找元素, 而无法按非关键字查找元素。

**例题 8-4** 集合 (同例题 8-2)。

分析:

根据上一章的分析, 只要查找 B 中的元素和 A 之间的关系即可。因此可以用散列表进行查找工作。首先将集合 A 进行散列存储, 哈希函数可以选择用除留余数法, 用链表进行存储。然后对 B 中的元素一一判断即可。

## 8.5 树表查找

树表查找是对树型存储结构所做的查找。树型存储结构是一种多链表，该表中的每个结点包含有一个值域和多个指针域，每个指针域指向一个后继结点。树型存储结构和树型逻辑结构是完全对应的，表示上都是一个树形图，只是用存储结构中的链接指针代替逻辑结构中的抽象指针罢了。因此，往往把树型存储结构（即树表）和树型逻辑结构（即树）混为一谈，统称为树结构或树。在本节的叙述中，对树的查找，就指对树表的查找。

在树结构中，有几种树对于数据的组织是非常有用的。即在第六章讨论过的二叉排序树和将在本章中介绍的堆和线段树。把一批数据组织成这两种树的形式往往比组织成线性表的形式更有效，即便于进行插入、删除和查找运算。在第六章中我们已经介绍了二叉排序树的插入、删除和查找运算，这里就不再重复。不过，二叉排序树有一个缺点，那就是树的结构事先无法预料，随意性很大，它只与结点的值和插入次序有关，往往得到的是—棵很不“平衡”的二叉树。二叉排序树与理想平衡树相差越远，树的高度就越高，其运算时间就越长。在最坏的情况下，就是对单链表进行运算的时间，从而部分或全部地丧失了利用二叉排序树组织数据的优点。为了克服二叉排序树的这个缺点，需要在插入和删除结点时对树的结构进行必要的调整，使二叉排序树始终处于一种平衡的状态，即始终成为—种平衡二叉树（balanced binary tree），简称平衡树。当然它不是理想平衡树，因为那将使调整操作更为复杂，使调整带来的好处得不偿失。

### 一、AVL 树

AVL 树（AVL tree）最先由俄罗斯两位数学家 Adelson - Velskii 和 Landis 在 1962 年提出来的，AVL 就是两人名字的首字母缩写。AVL 树近似于—棵理想平衡树，在插入和删除元素后能通过局部调整保持平衡。

AVL 树有些时候又被叫做“高度平衡树”，因为它基于保持每个结点左右子树高度平衡基础上保持整体平衡的。更正式的 AVL 树的定义如下：

—棵二叉搜索树是 AVL 树，当且仅当对于树中的每一个结点的左子树高度和右子树高度相差不超过 1。

假设把二叉搜索树的每个平衡因子定义为右树高度减去左树高度。如果—棵二叉搜索树为 AVL 树，那么树中平衡因子为 -1, 0 或者 1。如果存在一个结点不满足这个条件，那么二叉搜索树便不是 AVL 树。

图 8-6 中左边的二叉搜索树是—棵 AVL 树，而右边的不是。

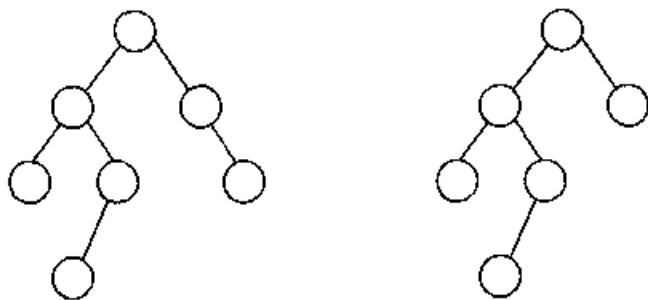


图 8-6 AVL 树比较示意图

对于许多平衡树来说,都有一个基本的操作,那就是树的旋转。这种树的左旋转和右旋转可以应用到任何一个二叉搜索树的结点上,使其变成一棵平衡二叉树。

图 8-7 就是一个在结点 25 上进行的右旋转操作。

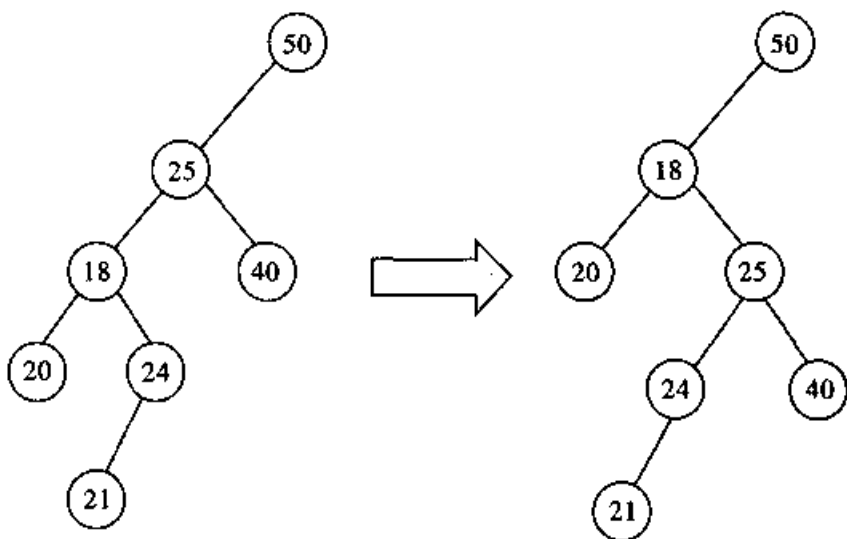


图 8-7 左旋转示例

在图 8-7 中对结点 25 进行右旋转操作:把 25 的左儿子 18 作为它的父亲结点,25 的右儿子不变。因为 25 是 18 的新右儿子,因此只能把 18 的右子树作为 25 的左子树。

因此右旋转和左旋转可以归结为图 8-8:

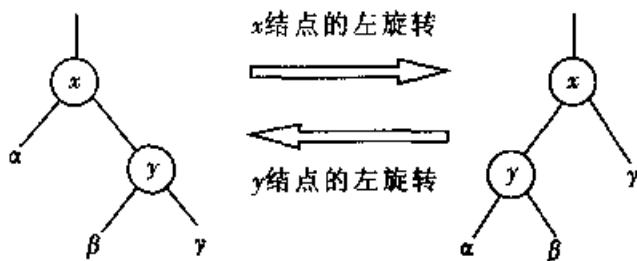


图 8-8 左右旋转示意图

接下来考虑 AVL 树的插入操作。

按照二叉搜索树插入结点的方法插入新元素,如果结果仍然是一个 AVL 树,那么直接退出。50% 都是这种情况。否则回溯从新结点到根的那一条路径,搜索那些结点平衡因子为 2 (或 -2),儿子平衡因子为 1 (或 -1)。如果儿子和父亲平衡因子正负符号相同,那么定义这种情况为 1 类调整,否则称作 2 类调整。

如果是 1 类调整,只要对平衡因子为 2 (或 -2) 的结点进行一次单独的旋转操作就可以了。如果平衡因子为 -2 那么进行右旋转。如果平衡因子为 2 则进行左旋转。这样能够保证这棵树仍是 AVL 树。

如果是 2 类调整,只要对儿子结点进行旋转,使儿子的平衡因子改变符号,这样 2 类调整变成了 1 类调整。然后进行 1 类调整,就可保证是一棵 AVL 树。然后继续回溯,遇到需要调整的结点进行调整,就能保证整棵树的平衡性了。

父结点平衡因子为 -2 的 1 类调整和 2 类调整的示意图如下:

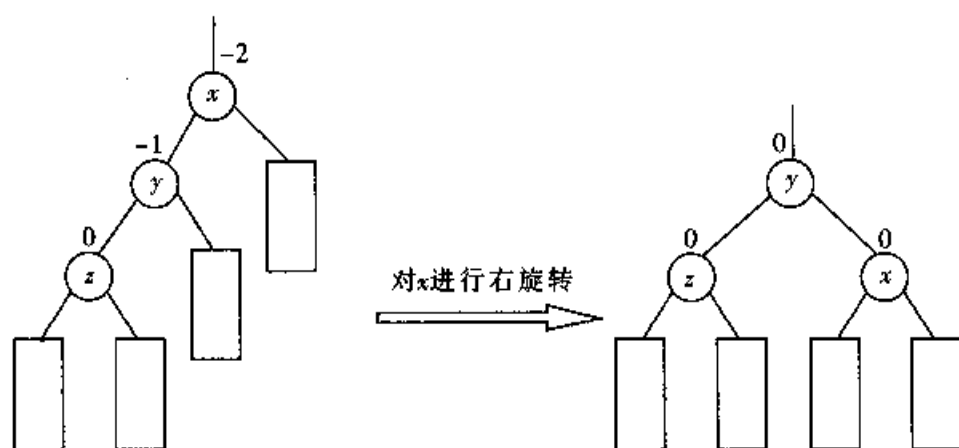


图 8-9 1 类调整

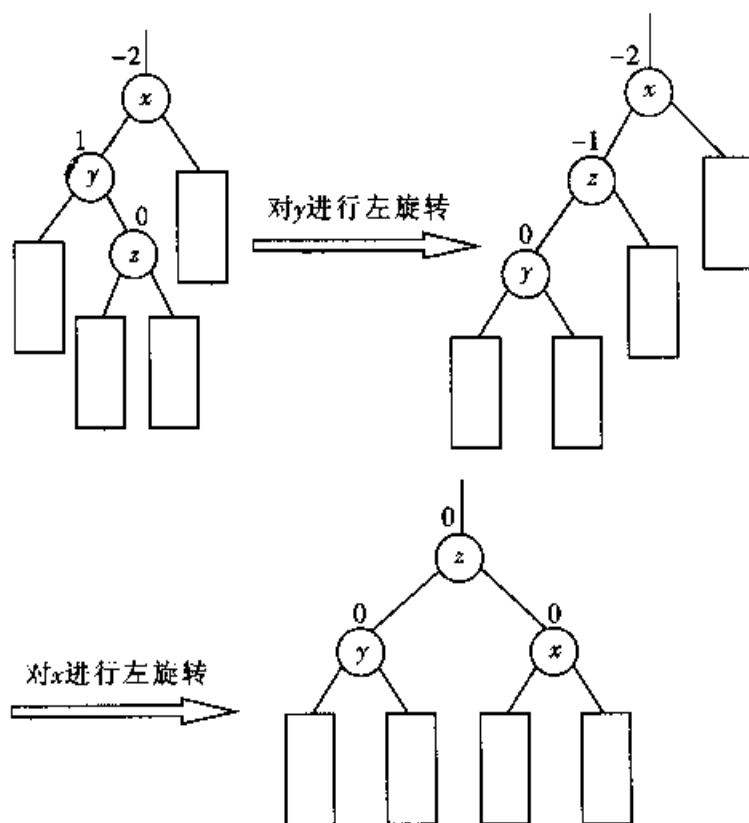


图 8-10 2 类调整

至于父结点平衡因子为 2 的情况相类似，只需要把左右方向翻过来即可，请读者自行画图思考。

可以看出 AVL 树的插入结点的复杂度与树的深度有关，因为两类调整的复杂度是常数级别，而 AVL 树的深度为  $O(\log_2 n)$ （证明略），所以总的时间复杂度为  $O(\log_2 n)$ 。

#### 例题 8-5 星星 (URAL 1028)。

问题描述：

宇航员经常检查星图，星图上面的星星用直角平面坐标系上的点表示。某个星星的级别等于

位于这个星星左下方的星星的个数。宇航员希望知道星图上每个星星的级别。如图 8-11，星星上的数字表示星星的级别。

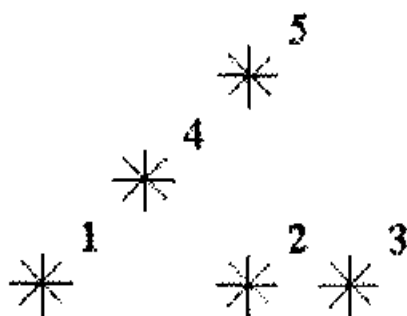


图 8-11 星图

现在给你星星的个数  $n$  ( $1 \leq n \leq 15000$ ), 和星星的坐标  $x, y$  ( $1 \leq x, y \leq 32000$ ), 请你求出每个星星的级别。

分析:

首先以星星的  $y$  坐标为关键字建立一棵 AVL 树, 然后将星星按照横坐标从小到大的顺序插入这棵 AVL 树。这样越靠左的星星越先插入这个 AVL 树。当要插入一颗星星的时候, 在这颗星星左边的都已经插入了 AVL 树, 因此只要统计出 AVL 树中有多少个结点的  $y$  坐标小于这颗星星, 那么这颗星星的级别就能统计出来了。

如何查找 AVL 树中有多少个结点小于某个结点, 只要对 AVL 树稍作扩充便可以了。在 AVL 中的每个结点扩充一个域  $ls$ , 记录这个结点的左子树结点的个数。插入某个元素的时候, 用  $count$  记录小于这个元素的结点个数。从根到插入位置 (AVL 未进行平衡调整之前的插入位置) 有一条路径, 对于这条路径上的结点  $i$ , 如果  $i$  的元素值小于插入元素, 那么  $count \leftarrow count + i.ls + 1$ 。因此在插入结点的过程中就将这个结点的级别计算出来了。

下面是一个统计过程的具体例子:

有如下星星:  $(0, 0), (1, 1), (2, 2), (2, 0), (3, 5)$ 。

按照星星的横坐标从小到大首先插入  $(0, 0)$ , 然后插入  $(1, 1)$ , 得到图 8-12 的 AVL 树。

然后插入  $(2, 2)$ , 在其插入路径上  $(0, 0), (1, 1)$  小于  $(2, 2)$ , 所以  $(2, 2)$  的级别加上  $(0, 0).ls + 1 + (1, 1).ls + 1 = 2$ , 得到下面的图 8-13:

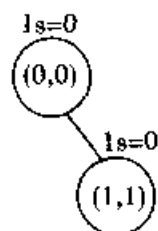


图 8-12 AVL 树

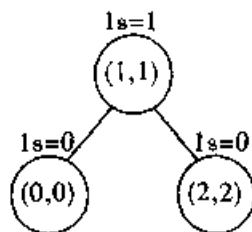


图 8-13 AVL 树

然后插入  $(2, 0)$ , 级别为 0。



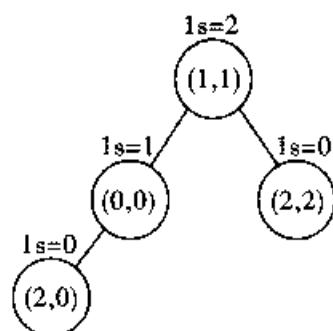


图 8-14 AVL 树

最后插入 (3, 5), 在其插入路径上, 有 (1, 1)、(2, 2) 小于它, 因此它的级别等于  $(1, 1) \wedge 1s + 1 + (2, 2) \wedge 1s + 1 = 4$ 。得到下图:

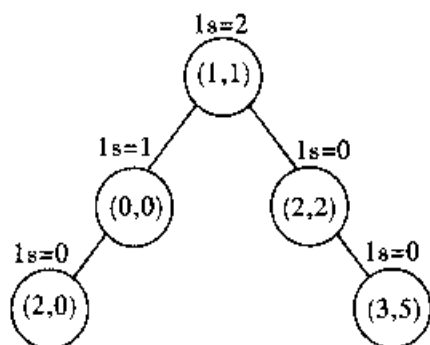


图 8-15 AVL 树

这样我们就求出了所有星星的级别 (0, 0) 为 0, (1, 1) 为 1, (2, 2) 为 2, (2, 0) 为 0, (3, 5) 为 3。

排序的复杂度为  $O(n \log_2 n)$ , 插入的复杂度为  $O(n \log_2 n)$ , 因此总的复杂度为  $O(n \log_2 n)$ 。

## 二、堆

堆 (heap) 是一种树型结构的优先队列, 通过树型结构维护根结点是树中最优先的结点的性质 (所谓最优先, 就是指在某种顺序关系下最优的, 比如最大或最小)。堆实质上是一棵满二叉优先级树。二叉优先级树与二叉搜索树稍有区别, 它是满足下面的优先性质:

- (1) 树中每一结点存储一个元素;
- (2) 树中任一结点中存储的元素的优先级高于其儿子结点中存储的元素的优先级 (也就是说高于其子树中任一结点的优先级)。

根据定义显而易见, 根是树中具有最高优先级的结点, 而且根到叶子任一条路径上结点的优先级从高到低。但这种优先级树和二叉搜索树一样会退化成一个线性表。由于在优先级树中执行插入或删除结点所需要的时间与树的高度有关, 所以最为合适的结构应该是一棵平衡的优先级树。当一棵优先级树是近似满二叉树时, 这棵树就叫做堆或偏序树。

堆的操作主要有三种: 建立堆、插入和删除结点。下面对这三种操作进行详细的介绍。

### 1. 插入结点

插入结点是基于一种形如“结点上浮”的方法。首先将新结点插入最底层的最右边, 如果最底层已满, 就插入下一层, 这样就保证了堆仍然是一棵近似满二叉树。但这样做可能破坏了堆的

优先性质。为了保持堆的优先性质，我们用一种结点上浮的方法：只要新元素的优先级高于其父结点，就交换两个元素的位置，直到新元素的优先级不高于其父结点或已经“上浮”成为根结点为止。下面就是一个插入结点的实例：

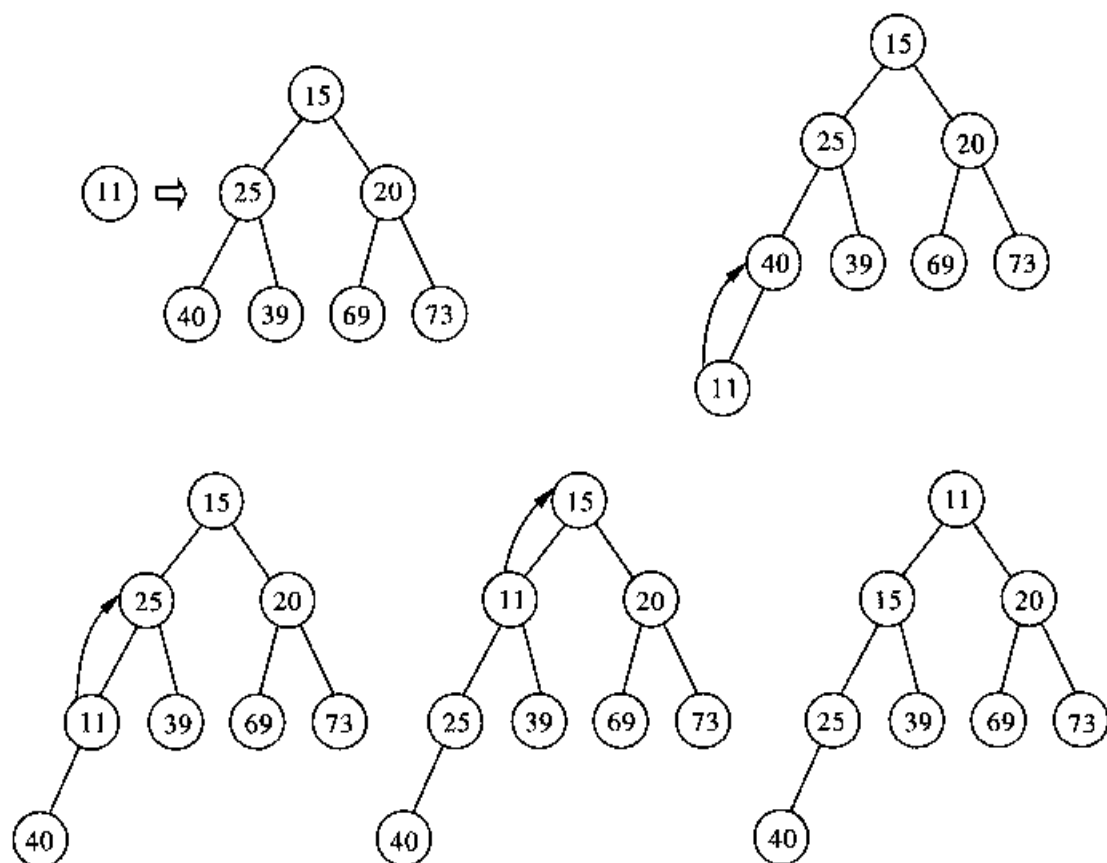


图 8-16 堆的插入结点过程

## 2. 删除结点

删除结点是基于一种形如“结点下沉”的方法。同样首先要保持堆的形态。删除一个结点时，如果它是最底层最右端的结点，那么堆仍然是一个近似满二叉树，否则就将最底层最右端的那个结点放置到被删除结点的位置，这样就保证了堆的形态不变。然后对那个结点进行位置的调整以维护优先性质：只要它的较高优先级的儿子结点的优先级高于此元素，就交换两个元素的位置，直到它的儿子的优先级都不高于它的优先级或者它已“下沉”到叶子的位置。下面是一个删除结点的实例：

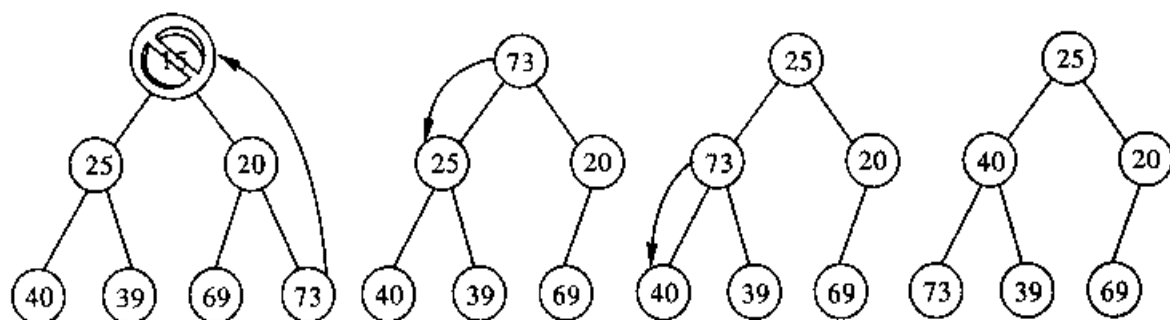


图 8-17 堆的删除结点过程

上述两种操作的复杂度都与树的高度有关，而堆是一棵近似满二叉树，因此高度最多为  $\log_2 n + 1$ ，因此复杂度为  $O(\log_2 n)$ 。

### 3. 建立堆

建立堆是指把一些元素按照优先级建立堆。很简单的想法是用插入结点来实现，现把一个结点看作一个堆，然后不断地把其他结点插入这个堆，直到所有结点都插进去，那么堆也就建成了。这样的建堆复杂度为  $O(n \log_2 n)$ ，似乎太慢了，下面介绍更好的方法。

首先按照任意顺序建立一棵包含所有待插入元素的近似二叉树，然后对其中的结点进行调整使其满足优先性质。从底向上逐次调整非叶子结点，首先检查第一个具有儿子结点，它有一个或两个儿子，如果以这个元素为根的子树已是最大堆，则此时不需调整，否则必须调整子树使之成为堆。然后检查它左边的非叶子结点等等，依次下去直到检查根结点，这样一个堆就建立成功了。下面是一个建堆的实例：

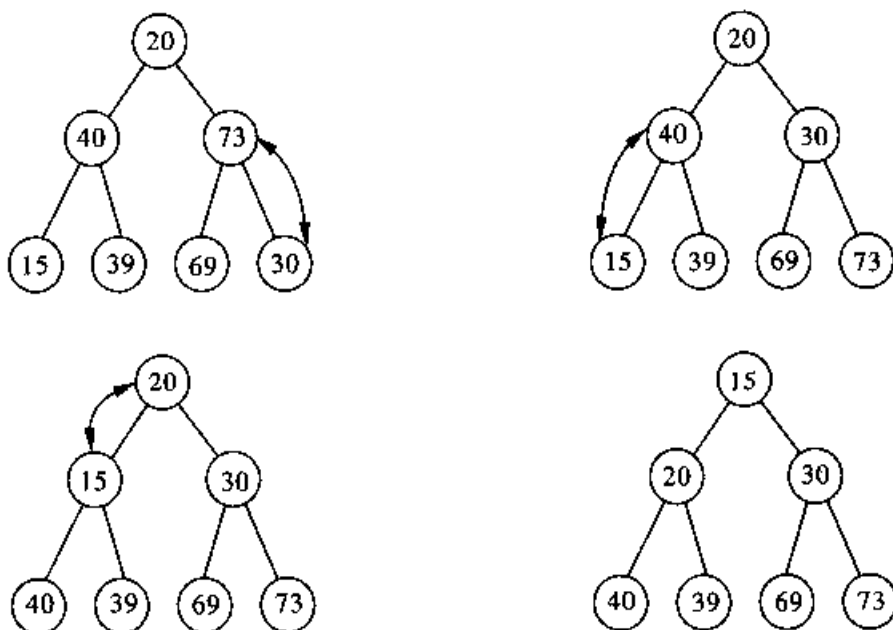


图 8-18 堆的建立过程

根据上面的算法简述，不难得出计算  $T$  的计算公式

$$T = \sum_{i=1}^n \frac{n}{2^{i+1}}$$

$$= n \sum_{i=1}^n \frac{1}{2^{i+1}}$$

$$\therefore T - \frac{T}{2} = n \left( \sum_{i=1}^n \frac{1}{2^{i+1}} - \sum_{i=2}^n \frac{1}{2^{i+2}} \right) \approx n \left( \frac{1}{4} - 0 \right)$$

$$\therefore T \approx n$$

$$\therefore O(T) = n$$

也就是说建立堆或者初始化堆的复杂度为  $O(n)$ ，这比我们开始所设想的  $O(n \log_2 n)$  快多了。

由于堆是一棵近似满二叉树，一般用二叉树的顺序存储对其进行程序实现。当堆中有  $n$  个元素时，可以将它放在一个数组  $H$  的前  $n$  个单元里。其中根结点中元素存放在  $H[1]$  中。一般地，

$H[i]$  的左儿子结点中的元素（如果存在）存放在  $H[2i]$  中； $H[i]$  的右儿子结点中的元素（如果存在）存放在  $H[2i+1]$  中。换句话说，当  $i > 1$  时， $H[i]$  的父结点中的元素存放在  $H[\lfloor \frac{i}{2} \rfloor]$  中。上述几种操作的伪代码实现如下：

#### 1. 插入结点

Procedure Insert ( $H, n, x$ )

$n \leftarrow n + 1$

$i \leftarrow n$

$j \leftarrow \lfloor i/2 \rfloor$

while ( $j > 0$ ) and ( $x$  优先于  $H[j]$ ) do

$H[i] \leftarrow H[j]$

$i \leftarrow j$

$j \leftarrow \lfloor i/2 \rfloor$

$H[i] \leftarrow x$

#### 2. 下沉结点

Procedure Down ( $H, n, x, i$ )

$j \leftarrow i \times 2$

while  $j \leq n$  do

if ( $j < n$ ) and ( $H[j+1]$  优先于  $H[j]$ )

then  $j \leftarrow j + 1$

if  $H[j]$  优先于  $x$

then

$H[i] \leftarrow H[j]$

$i \leftarrow j$

$j \leftarrow i \times 2$

else break

$H[i] \leftarrow x$

#### 3. 删除结点

Procedure Delete ( $H, n, i$ )

$n \leftarrow n - 1$

if ( $n = 0$ ) or ( $i = n + 1$ ) then exit

Down ( $H, n, H[n+1], i$ )

#### 4. 建立堆

Procedure Build ( $H, n$ )

for  $i \leftarrow \lfloor (n+1)/2 \rfloor$  downto 1 do

Down ( $H, n, H[i], i$ )

**例题 8-6** 给你  $n$  个元素，请你利用堆对其从小到大进行排序。（ $N \leq 10000$ ）

分析：首先将  $n$  个元素建立小根堆（就是根结点是最小元素）。然后进行根结点的删除，将每次删除的根结点按顺序记录下来，直到堆为空。这样保存下来的出堆序列就是按从小到大排序的。

建立堆的复杂度为  $O(n)$ ，删除结点的复杂度为  $O(n \log_2 n)$ ，所以总的复杂度为  $O(n \log_2 n)$ ，达到了基于比较的排序的复杂度理论下限。在实际应用中，堆排序的速度并没有快速排序快，但是速度非常平稳，没有最坏情况。堆排序是不稳定排序。

### 三、线段树

线段树 (interval tree) 又叫做区间树，它的形态是一个静态的区间集合，又由于在计算几何中广泛的应用，所以被称为线段树。区间有闭区间、开区间、半开半闭区间，在这一节中假设讨论的区间都是闭区间，就是形如  $[t_1, t_2]$  ( $t_1 \leq t_2$ ) 的实数对，表示集合  $\{t \in \mathbb{R}; t_1 \leq t \leq t_2\}$ 。

我们可以将区间  $[t_1, t_2]$  定义为一个线段类型，那么线段类型  $i$  有两个域： $\text{low}[i]$  和  $\text{high}[i]$ 。其中  $\text{low}[i]$  表示下端点，即  $\text{low}[i] = t_1$ ； $\text{high}[i]$  表示上端点，即  $\text{high}[i] = t_2$ 。我们说两个线段  $i$  和  $i'$  相交，当且仅当  $i \cap i' \neq \emptyset$ ，即就是  $\text{low}[i] \leq \text{high}[i']$  且  $\text{low}[i'] \leq \text{high}[i]$ 。任意两条线段  $i$  和  $i'$  满足下列分类：

- $i$  和  $i'$  相交
- $i$  在  $i'$  的左边，即  $\text{high}[i] < \text{low}[i']$
- $i$  在  $i'$  的右边，即  $\text{high}[i'] < \text{low}[i]$

一棵线段树是一棵二叉搜索树维持着一个静态线段集合，其中的每一个结点  $x$  都包含了一条线段  $\text{int}[x]$ 。设  $\text{int}[x] = [t_1, t_2]$ ，那么结点  $x$  的两个儿子结点  $x.l$ ,  $x.r$  包含的区间分别为  $\text{int}[x.l] = [\lfloor \frac{t_1+t_2}{2} \rfloor, t_1]$ ， $\text{int}[x.r] = [\lfloor \frac{t_1+t_2}{2} \rfloor, t_2]$ 。最终线段树的每一个叶子结点的区间形为  $[i, i+1]$ 。每个结点  $x$  还有一个域  $\text{count}[x]$  表示覆盖该结点的线段条数。线段树图 8-19 如下：

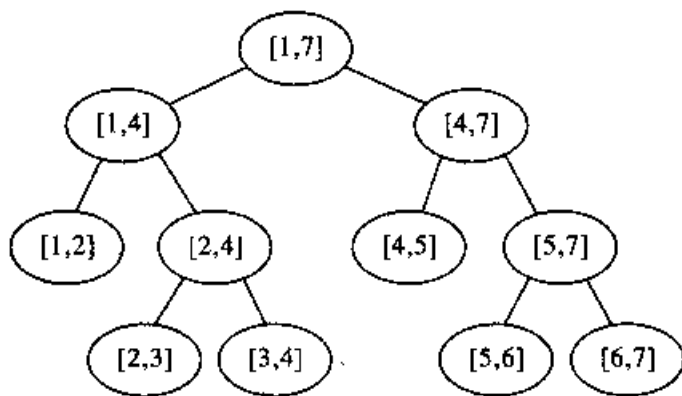


图 8-19 线段树示意图

线段树的优势是：当求解某一个结点  $x$  区间为  $[t_1, t_2]$  的时候，可以直接查找这个结点  $x$ ，而不要从  $[t_1, t_1]$  到  $[t_2, t_2]$  一个个累加。这是线段树在时间效率上主要的优点。线段树是一个动态维护的数据结构。考虑空间复杂度，最坏情况是满二叉树，结点总数为  $L + \lfloor L/2 \rfloor + \lfloor L/4 \rfloor + \dots = 2L$  ( $L = b - a$ )。

线段树的插入和删除算法是基于二分的。只要考虑的总区间不变，线段树的结点本身是不会改变的，改变的只是计数器和其他信息。线段树的插入和删除操作是二分的，而且可能要同时递归到两个分支中。但是任意层中最多只有两个未被完全覆盖的区间需要递归处理，因此插入和删

除的复杂度为  $O(h)$ ，其中  $h$  为树的高度（这可以数学归纳法证明）。线段树的插入和删除代码如下：

```

Procedure InsertIntervalTree (x, l, r, c)
  if (l ≤ low [x]) and (high [x] ≤ r)
    then count [x] ← count [x] + c
  else
    mid ← (low [x] + high [x]) div 2
    if l ≤ mid then InsertIntervalTree (x.l, l, mid, c);
    if mid ≤ r then InsertIntervalTree (x.r, mid, r, c);
    
```

除了完全覆盖区间的线段条数  $\text{count}[x]$  之外，往往还需要记录两个量来帮助我们获得更多的信息：

测度  $m$ ：结点所表示区间中线段覆盖过的长度。

独立线段数  $\text{line}$ ：指的是区间中互不相交的线段条数。

权和  $\text{sum}$ ：区间所有元线段的权和。

由于线段树是一棵近似满二叉树，因此其结点数的级别等于叶子结点个数的级别。由于叶子结点的个数等于单位线段的个数，若线段树表示的线段范围是  $1 \sim N$ ，则线段树空间复杂度为  $O(n)$ 。

#### 例题 8-7 线段染色 (URAL 1019)。

问题描述：

有一条很长的整数数轴，它的范围是  $0 \sim 10^9$ 。然后将其中的一些线段染成白色，又有一些线段又重新被染成黑色。这里总共被  $N$  ( $1 \leq N \leq 5000$ ) 次染色。问反复染色完毕后，最长的白色线段有多长？

输入：

第一行为  $N$ ，接下来  $N$  行为染色的描述  $l, r, c$ ：表示从  $l$  到  $r$  染成  $c$  这种颜色， $c$  值是 ‘b’ 表示染成黑色，‘w’ 表示染成白色。

输出：

最长的白色线段。

输入样例：

```

4
1 999999997 b
40 300 w
300 634 w
43 47 b
    
```

输出样例：

```

47 634
    
```

分析：

这是一道很典型的线段树题目。由于数轴的范围是  $0 \sim 10^9$ ，不能直接根据这个数值范围构造线段树。然而只有  $n$  次染色，因此可以将线段端点离散，利用这些端点建立线段树。所谓离散，

就是将上述端点值进行从小到大排序, 然后对每个数值重新标号。例如题目中的样例, 将端点排序后: 1, 40, 43, 47, 300, 999999997, 给这些端点从 1 至 6 重新标号, 这样新端点 (或者叫映射点) 1 对应 1 (记作  $p[1] = 1$ ), 映射点 2 对应 40 ( $p[2] = 40$ ), 映射点 3 对应 43 ( $p[3] = 43$ ) ……这样新端点 (映射点) 保持了连续性, 且映射点的个数是  $O(n)$  级别的, 只是相邻映射点对应的单位线段长度不相等。可见离散之后, 线段树的复杂度和坐标范围没有关系。

端点离散后, 将这些新端点构成的线段建立线段树。线段树中的每个结点除了常规域, 还增加了一个域: `cover`。`cover` 记录的是这个结点对应线段是否全部被染成了白色或者黑色, 否则有黑有白。通过 `cover` 域可以判断一个单位线段是什么颜色, 这样就可以方便后面的统计。

接下来, 看如何对 `cover` 进行更新。插入线段时, 若一个结点完全被插入的线段包含, 那么只需要将此结点 `cover` 标记一下, 而不需要再更新其子结点。若插入的线段只是结点的一个子线段, 就需要分两种情况考虑: 若这个结点是单色结点 (也就是被一个单色线段完全覆盖), 那么在线段插入子结点之前, 将其子结点标记为单色结点 (因为在上一情况中, 没有标记单色结点的子结点)。若这个结点不是单色结点 (也就是有黑色也有白色), 那么直接继续更新其子结点。这样每次维护结点只需要  $O(\log_2 n)$  的复杂度。

所有的染色任务完成之后, 就可以确定每一条单位线段的颜色了。枚举每一条单位线段, 在线段树中查找它的颜色。在树上查找时, 如果存在一个单色结点的线段包含要查找的线段, 那么要查找的线段的颜色就和这个线段相同。这样对每条单位线段查找的复杂度为  $O(\log_2 n)$ , 而单位线段的级别为  $O(n)$ , 所以这一步总的复杂度为  $O(n \log_2 n)$ 。确定了每条单位线段的颜色之后, 只要对其扫描一遍, 找出最长连续白色线段即可。

通过上面的分析可知, 算法时间复杂度为  $O(n \log_2 n)$ , 而空间复杂度为  $O(n)$ 。

## 8.6 查找的应用举例

查找是算法设计中最基本的一步, 几乎在所有的算法设计中都牵涉到了查找算法。查找算法只有在和其他算法的结合下才能发挥它最为强大的作用。下面结合实例介绍查找算法的广泛应用。

**例题 8-8 书稿复制 (cerc98)。**

**问题描述:**

有  $n$  本书 ( $1 \leq n \leq 10000$ ), 编号  $1, 2, \dots, n$ 。每本  $p_i$  页。全部分给  $m$  个抄写员。每人分到顺序连续的若干本, 每本只分给一人。求一种方案, 使每人分到的页数和的最大值为最小。

**例子:**  $n=9, m=3$

100 200 300 400 500 / 600 700 / 800 900

**分析:**

看到这个题目, 有的同学可能想到用动态规划解决。事实上由于这个题目的特殊性, 用二分和贪心是一种更为简单巧妙的方法。

首先可以肯定一点, 假若每人分到的页数和的最大值  $\max$  一定, 那么按照序列从头到尾分配

页数给同一个人的时候，一定是尽量选取多的书给这个人，只要页数之和小于  $\max$  就可以了。然后可以推出，若这样分配的方案中得到的实际最大值  $\max'$  小于假定的最大值  $\max$ ，那么就说明最优方案仍有可能调整，于是将  $\max$  减小，再用上面的贪心法就可以判断是否是最优方案。若无法将书分配完，说明  $\max$  太小，那么将  $\max$  增大再用贪心法即可。若实际最大值  $\max' = \max$ ，那么说明  $\max$  就是最优方案了。怎样调整  $\max$  呢？明显发现， $\max$  是单调的，说明我们可以用二分法！伪代码如下：

```
l ← 1
r ←  $\sum p_i$ 
while l < r do
    mid =  $\lfloor \frac{(l+r) + 1}{2} \rfloor$ 
    if 用贪心法设置  $\max = \text{mid}$  是可行的
        then  $l \leftarrow \text{mid}$ 
    else  $r \leftarrow \text{mid} - 1$ 
```

源程序：

```
program ex8-8;
const
    inf = 'input.txt';
    outf = 'output.txt';
    maxn = 10000;
var n, m; longint;
    p: array [1..maxn] of longint;
function check (k: longint): boolean;
var t, now, i; longint;
begin
    t := m; now := k;
    for i := 1 to n do if p[i] ≤ now then now := now - p[i]
    else begin
        t := t - 1; now := k - p[i];
        if t = 0 then break;
    end;
    check := (t = 0) or (t = 1) and (now = 0);
end;

var i, l, r, mid; longint;

begin
    assign (input, inf); assign (output, outf);
    reset (input); rewrite (output);
```



```

read (n, m);
l := 0; r := 0;
for i := 1 to n do begin
  read (p[i]); r := r + p[i];
  if l < p[i] then l := p[i];
end;
while l < r do begin
  mid := (l + r + 1) shr 1;
  if check (mid) then l := mid
  else r := mid - 1
end;
writeln (l);
close (input); close (output);
end.

```

### 例题 8-9 彩色项链。

#### 问题描述：

一条项链由  $N$  个珠子连接而成，编号依次为  $0, 1, 2, \dots, N-1$ 。每个珠子的颜色用  $0 \sim 9$  之间的一位数字来表示（因此，可用的颜色一共有 10 种）。一条长度为 4 的项链如图 8-20 所示：（圆圈中的数字表示颜色，圆圈旁边的数字为珠子的编号）

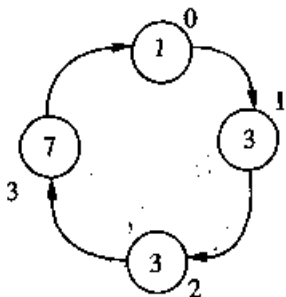


图 8-20 一条长度为 4 的项链

需要注意的是，编号为  $0, 1, 2, \dots, N-1$  的珠子大小是依次递增的，设编号为  $i$  的珠子的颜色值为  $a_i$ ，则数字序列  $a_0 a_1 \dots a_{n-1}$  可以唯一地表示一种项链。例如，图 8-20 所示的项链表示为“1337”。

现在有一台自动生产项链的机器，它的结构和工作方式如下所述：

机器的核心控制部件主要包括：一个 CPU、一个整数寄存器 START 和存储器 S。

机器内部固化有一段程序，由 CPU 解释执行。该程序的输入是长度为  $N$  的十进制数字序列 A，输出是另一个长度为  $N$  的十进制数字序列 B。每次执行程序前将 S 初始化为输入序列 A，程序结束后把 S 作为输出串 B。START 初始化为 0。

程序包含  $M$  条指令，顺序编号为  $1 \sim M$ 。指令共有 5 种，以下是指令的格式和功能：（尖括号



< > 表示指令参数，都是整数)

编号	功能	格式	说明
1	设置寄存器	SETSTART <a> <b>	设置 START 的值：从 S 的第 a 位开始取连续 b 位得到的十进制整数（可能大于 N-1）。 $0 \leq a \leq N-1, 1 \leq b \leq \min(5, N)$
2	循环移位	SHIFT <L> <x>	把 S 从第 START 位开始的连续 L 位，循环移位 $ x $ 位。 $x > 0$ 时右移， $x < 0$ 时左移。 $2 \leq L \leq \min(10, N), 1 \leq  x  \leq L-1$
3	乘法	MUL <L> <x>	把 S 从第 START 位开始的连续 L 位当做原始串（L 位十进制整数），将其乘以 x 以后保留结果的最低 L 位，替换原始串。 $1 \leq x \leq 9, 1 \leq L \leq \min(10, N)$
4	条件	ONDIGIT <x> <y> <z>	如果 S 的第 x 位等于 y，则跳转到第 z 条指令。 $0 \leq x \leq N-1, 0 \leq y \leq 9, 1 \leq z \leq M$
5	终止	END	仅作为最后一条语句出现，程序终止。

由于项链是环型的，因此第 i 位和第  $i+kN$  位（k 为整数）代表数字序列的同一位置。例如当  $N=4$  时，第 6 位和第 2 位是等价的。

下面是一个程序的例子：

```
MUL 3 2
SETSTART 2 1
ONDIGIT 0 4 1
SHIFT 3 -2
END
```

机器启动的时候，输入一个数字串  $S_0$ ，执行程序得到一个新的数字序列  $S_1$  并生产出  $S_1$  代表的项链来，以后机器每生产出一条新项链  $S_n$ ，就把  $S_n$  对应的数字序列作为输入重新执行一遍程序，得到一个新的数字序列  $S_{n+1}$  并生产出新的项链。

由于长度为 N 的项链种类数目是有限的（至多  $10^N$  种不同的项链），因此如果让机器一直工作

下去,某些种类的项链会被生产出无限多条。编程计算出这些将被无限生产出的项链有多少种。在本题中,可以被生产出来的项链种类总数保证不超过  $10^6$ 。

分析:

将这道题目抽象出一个数学模型:给定一个初始状态  $S_0$  和一个函数  $f(T)$  {状态为一个长度为  $N$  的 10 进制整数,函数  $f(T)$  的返回值为另一个状态或者  $T$  本身},  $i \in \mathbb{Z}^+$  时  $S_i = f(S_{i-1})$ 。求无限序列  $S$  中出现了无限次的状态个数。其中函数  $f(T)$  就是制造项链的程序。

把每个状态用一个点表示,每个状态  $S$  对应的点连一条弧到  $f(S)$  所对应的点。从  $S_0$  开始,不断沿着唯一的出弧往下走,问题就是求会被走过无限次的点的总数。经过简单的分析就会知道,这条无限长的路线分为两个部分——一条链  $A$  和一个圈  $B$  (如图 8-21),  $B$  中的点将被访问无限次,不在  $B$  中的点只可能被访问到 0 次或者 1 次,因此问题就是求  $B$  中点的个数(称之为  $B$  的长度)。

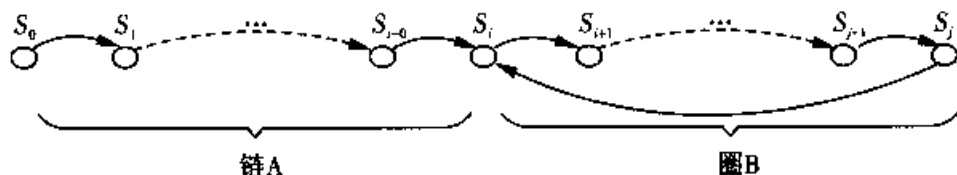


图 8-21 路线示意图

因为数据规模的庞大,我们不能把产生的所有状态都保存下来。但有一个基本的思想便是:找到位于圈  $B$  上的某一个状态作为标志保存下来,然后沿着这个状态走下去,直到再次找到这个点,那么就确定了圈的长度。关键是要确定圈上的那个标志,方法主要有下面两种。

如何求  $B$  的长度呢?有两种初始的想法:

一、从  $S_0$  开始逐个产生  $S_1, S_2, \dots$ , 每产生一个  $S_j$ , 就查找以前是否已经产生了一个状态  $S_i$  使得  $S_i = S_j$ , 如果找到就输出  $j - i$ 。

但是考虑到数据的范围:  $N \leq 10^5, j \leq 10^6$ , 空间上是无法承受的:因为要检查  $S_i = S_j$ , 就必须把以前产生的所有状态存下来,需要  $10^{11}$  Byte, 即使用再好的方法压缩,也不可能用 64MB 存储下来。时间上如果不用优化更加是无法承受的。(不过,虽然直接这样做不行,但是利用这种思路还是能够很高效地解决本题的,下文将会做详细的分析)

二、找到圈  $B$  上任意一点  $i$ , 设对应状态为  $S_i$ , 从  $S_i$  开始逐个产生  $S_{i+1}, S_{i+2}, \dots$ , 直到产生某个状态  $S_j$ , 使得  $S_j = S_i$ 。

但是实现起来面临着两个新问题:①如何求找到  $B$  上的任意一点;②如何判断状态  $S_j$  和  $S_i$  相等。

首先来解决问题①,有两种方法:

1. 由已知条件“不同状态总数不会超过  $10^6$ ”可知  $S_0^6$  肯定在圈上,因此可以令  $i = 10^6$ ;但是计算  $S_0^6$  要计算  $10^6$  次函数  $f$  的值,也就是模拟运行  $10^6$  次给定的小程序,速度是非常慢的;

2. 令  $C_0$  为  $S_0, C_i = f(f(C_{i-1}))$ , 易知  $C_i = S_{2i}$  可以证明一定存在某个  $i$  使得  $C_i = S_i$ , 且这个  $C_i$  一定在圈  $B$  中。证明如下:

$C$  序列可以看作某个人 Person1 从  $S_0$  对应顶点开始每次往下沿着唯一的出弧走两步的访问顶点序列,而  $S$  序列则是另一个人 Person2 每次走一步的访问顶点序列。当 Person2 第一次进入圈  $B$  中时, Person1 肯定已经在  $B$  中了,设 Person1 此时的状态为  $C_{N_{\text{new}}}$ , Person2 此时的状态为  $S_{N_{\text{new}}}$ , 由

于  $C_{Now}$ 、 $S_{Now}$  都在圈 B 中，所以  $C_{Now}$  经过若干次  $f$  变换一定能够变为  $S_{Now}$ ，即，那么两人再走  $L$  次 Person1 就能够“追”上 Person2。因为，易知  $C_{Now} + L$  也在圈 B 上，令  $i \leftarrow Now + L$  则命题得证。

根据上面的证明过程可知，若链 A 的长度为  $L_A$ ，圈 B 的长度为  $L_B$ ，那么 Person1 和 Person2 总共走  $L_A + L_B$  次就一定能够达到相同的顶点，而一旦走到同一个顶点就可断定这个点对应的状态在圈 B 中。

不过方法 2 中又需要判断两个状态  $C_{Now}$  和  $S_{Now}$  是否相等的问题，不过这和问题②是十分类似的，而问题②将在下文给出具体解法，所以这里不再赘述。

在大多数情况下， $L_A + L_B < 10^6$ ，但是在最坏情况下（ $L_A = 0$ ， $L_B = 10^6$ ），计算函数  $f$  的次数达到  $3 \times 10^6$ ，是方法 1 的 3 倍，所以对于不同的测试数据方法 1 和方法 2 的效率各有千秋。当然也可以把两个方法结合起来，在方法 2 的基础上加一条判断语句：if  $i = 10^6/2$  then break；这样最坏情况下只需计算  $3 \times 10^6/2 = 1.5 \times 10^6$  次  $f$  函数（不过对于所有标准测试数据，作用都不很明显）。

再来看一看问题②：这个问题最简单的方法为  $O(N)$  的循环判断每一位是否相同，但是这样做的复杂度太高了，时间上无法承受，所以我们要利用题目的另一些条件来减少运算次数。根据题目：计算一次  $f$  函数实际上是几次循环位移或者乘法操作，我们可以在模拟这些操作的同时，完成对两个状态的比较。具体方法如下：

设要计算状态  $S$  的  $f$  值，Same 为状态  $S$  与目标状态  $T$  的相同位置的个数，当  $Same = N$  就表示状态  $S$  与  $T$  是完全相同的。在进行一次循环位移或乘法操作时，Same 的值发生改变：若  $S$  中的第  $i$  位 ~ 第  $j$  位变为了  $X_i \sim X_{j-1+i}$ ，则  $Same \leftarrow Same + \sum_{k=1}^j [\text{Ord}(X_{k-1+i} - T_k) + \text{Ord}(S_k - T_k)]$ 。可以看出：Same 的计算只是在模拟的同时进行的，只是增大了复杂度中的系数，并没有增加任何时间复杂度。

下面给出方法 2 的大致框架：

1. 输入；

2. 找到圈 B 上的某个点  $s1$ ：

same : = n;  $s1$  : =  $s0$ ;  $s2$  : =  $s0$ ; count : = 0;

repeat

work ( $s1, s2$ ); work ( $s1, s2$ ); work ( $s2, s1$ ); inc (count, 2);

until (same = n) or (count  $\geq$  1000000);

{ Work ( $a, b$ ) 的作用：将  $a$  的下一个状态赋给  $a$ ，同时修改  $a$  与  $b$  的相同位置的个数 }

3. 计算圈 B 的长度：

count : = 0;  $s2$  : =  $s1$ ; same : = n;

repeat

work ( $s1, s2$ ); inc (count);

until same = n;

6. 输出 count;

注：此算法是出题者提供的标准方法（称之为算法 1），但并不是最优的，利用第一种思想可以得到一个准确率极高而且效率很高的算法（称之为算法 2），下面给出分析。

我们知道：前面提到的第一种思路中最难处理的一步就是查找以前是否已经产生了一个状态  $S_i$  使得  $S_i = S_j$  {包括保存状态}。假如我们对每个状态取 mod 值，mod 一个数  $P$  ( $P > N$ )，如果它们 mod  $P$  值相同则认为它们是同一个状态。这样做不会误把两个不同的状态当成相同的状态的概率有多大呢？

容易知道：这等于把  $N$  个小球放入  $P$  个盒子中使得任两个小球都不在同一个盒子中的概率：

显然等于  $\frac{\text{任意小球都不在同一个盒子中的方案数}}{\text{总的方案数}} = \frac{C_P^N N!}{P^N}$ ，而  $\frac{P!}{(P-N)!} = \frac{P}{P} \cdot \frac{P-1}{P} \cdot \frac{P-2}{P} \cdots \frac{P-N+1}{P} > \left(\frac{P-N+1}{P}\right)^N$ ，当  $P$  足够大时，概率已经十分接近 1 了，比如当  $P$  等于  $10^{16}$  时，概率  $> 0.9999$ 。（当然我们做的时候不能取  $P$  等于  $10^{16}$ ，因为这样一来只要末 16 位数字相同就会认为两个状态相同，我们不妨取  $P = 10^{16} - 3$ ）

那么，我们比较两个状态时，可以直接比较它们 mod  $P$  是否相等。

我们保存一个状态时，无须存储一个状态所有的  $N$  位数字，而只需存下它 mod  $P$  的值就够了，换言之可以用一个  $0 \sim P-1$  的整数代表一个状态。而查找则可以用静态 Hash 表来做，查找复杂度为常数，如果 Hash 函数为 mod ( $10^6$  左右的一个质数) 则常数大致为 1。而计算一个长度为  $N$  的长整数  $X$  mod  $P$  的值可以按下列方法：设此数为  $X_{N-1}X_{N-2}\cdots AX_1X_0$ ，设  $V_i = 10^i \text{ mod } P$ ，则  $X \text{ mod } P = (\sum_{i=0}^{N-1} X_i V_i) \text{ mod } P$ ，但是同样这个复杂度也太高，所以可以类似问题②的解决方法，在修改状态时对 mod 值做出相应的调整。

具体的程序由读者自己去完成。

### 例题 8-10 营业额统计。

问题描述：

一个公司需要统计自公司成立以来的营业情况。营业情况是这样统计的：由于在不同时候，公司的营业额会出现一些波动，当波动很大时，公司的经营状况就出现了问题。因此，经济管理学商定义了一种最小波动值来衡量这种情况：

当天的最小波动值 =  $\min \{ | \text{当天的营业额} - \text{以前某一天的营业额} | \}$

{第一天的最小波动值 = 第一天的营业额}

你的任务就是，对公司成立以来  $n$  ( $n \leq 32767$ ) 天的营业额  $a_i$  ( $a_i \leq 1000000$ ) 进行分析，把每天的最小波动值加起来。

分析：

通过上面的问题描述，发现问题的本质是：有一整数序列  $\{a_n\}$ ，对于这个序列的每个元素  $a_i$  ( $1 \leq i \leq n$ )，找一个  $a_j$  ( $j \leq i$ ) 使其最为接近  $a_i$  (即  $|a_i - a_j|$  最小)，再把这些差值累加起来。可以表示为下列式子：

$$s = \sum_{i=1}^n \sum_{j=1}^{i-1} \min (|a_i - a_j|)$$

问题的关键便是怎样快速地查找每一个元素  $a_i$  所对应的  $a_j$ 。如果用顺序查找，那么查找时间复杂度为  $O(n)$ ，总时间复杂度为  $O(n^2)$ ，显然是无法承受的。想到这章我们所学的平衡二叉排序树，用它进行的查找复杂度只需  $O(\log_2 n)$ 。

利用平衡二叉树很简单，将每天的营业额作为二叉树的结点搭建一棵平衡二叉树。然后按照时间先后将营业额  $a_i$  插入这棵平衡二叉树。在插入  $a_i$  的过程中，有一个从根到目标位置的路径，可以证明与  $a_i$  最接近的元素一定位于这条路径上。因此将这些差值累加起来就是最后的答案了。而每次插入元素的复杂度为  $O(\log_2 n)$ ，因此总的复杂度为  $O(n \log_2 n)$ 。

源程序：

Program ex8 - 10;

const

fl = 'input.txt';

f2 = 'output.txt';

maxn = 100000;

type

integer = longint;

rec = record

left, right, ld, rd, data, h: integer;

end;

var

a: array [1..maxn] of rec;

mm, n, len, i, root: integer;

w, ans, ans2: int64;

function min (i, j: integer): integer;

begin

if i < j then min := i

else min := j;

end;

procedure ins (var k: integer; fa: integer);

var rs, ls: integer;

begin

if k = 0 then begin

k := len;

if w < a[fa].data then begin

ls := a[fa].ld;

a[fa].ld := len;

a[len].rd := fa;

a[len].ld := ls;

if ls < > 0 then a[ls].rd := len;

if ls = 0 then inc (ans, abs (a[fa].data - a[len].data))

else inc (ans, min (abs (a[fa].data - a[len].data), abs (a[ls].data - a[len].data)));

end else begin

```

    rs := a[fa].rd;
    a[fa].rd := len;
    a[len].ld := fa;
    a[len].rd := rs;
    if rs < > 0 then a[rs].ld := len;
    if rs = 0 then inc(ans, abs(a[fa].data - a[len].data))
    else inc(ans, min(abs(a[fa].data - a[len].data), abs(a[rs].data - a[len].data)));
end;
exit;
end;
mm := min(mm, abs(a[k].data - w));
if w < a[k].data then begin
    ins(a[k].left, k);
    ls := a[k].left;
    if a[ls].h > a[k].h then begin
        rs := a[ls].right;
        a[k].left := rs;
        a[ls].right := k;
        k := ls;
    end;
end else begin
    ins(a[k].right, k);
    rs := a[k].right;
    if a[rs].h > a[k].h then begin
        ls := a[rs].left;
        a[k].right := ls;
        a[rs].left := k;
        k := rs;
    end;
end;
end;
procedure show(t: integer);
begin
    if a[t].left < > 0 then show(a[t].left);
    writeln(a[t].data);
    if a[t].right < > 0 then show(a[t].right);
end;
begin

```

```

randomize;
assign (input, f1); assign (output, f2);
reset (input); rewrite (output);
readln (n);
ans: =0; ans2: =0;
len: =0; root: =0;
for i: =1 to n do begin
  if eof (input) then w: =0
  else readln (w);
  mm: = maxlongint;
  inc (len);
  a [len] .data: = w; a [len] .left: =0; a [len] .right: =0;
  a [len] .h: = random (maxint); a [len] .ld: =0; a [len] .rd: =0;
  if root < > 0 then begin ins (root, 0); inc (ans2, mm); end
  else begin
    root: = 1;
    inc (ans, w);
    ans2: = w;
  end;
end;
writeln (ans); writeln (ans2);
close (input); close (output);
end.

```

### 例题 8-11 最轻的语言。

问题描述:

$A_k$  字母表由英语字母表中最初的  $K$  个字母组成。称为重量的正整数分别代表每个字母表中的字母重量。来自于  $A_k$  字母表中字母组成的单词重量等于该单词中所有字母重量的总和。关于  $A_k$  字母表的语言是由该字母表组成的任何有限的单词。语言的重量是其所有单词重量的总和。如果该语言中每对不同的单词  $W$ 、 $V$ ， $W$  不是  $V$  的前缀，那么我们就说该语言是无前缀的。

我们想找出关于字母表  $A_k$  的  $n$  元素的无前缀的语言最轻的重量是多少。

例如:

假定  $K=2$ ，字母  $a$  的重量—— $W(a)=2$ ，字母  $b$  的重量—— $W(b)=5$ 。

单词  $ab$  的重量—— $W(ab)=2+5=7$ 。 $W(aba)=2+5+2=9$ 。语言  $J=\{ab, aba, b\}$  的重量—— $W(J)=21$ 。语言  $J$  不是无前缀的，因为单词  $ab$  是  $aba$  的前缀。关于字母表  $A_2$  的最轻的 3 元素的，无前缀的语言（假定字母的重量依据前面所给的）是  $\{b, aa, ab\}$ ，它的重量是 16。

任务：编写一个程序，计算关于  $A_k$  字母表的  $n$  元素的无前缀的语言的最轻重量。



分析:

对于字母表中得  $k$  个字母, 任意交换两个字母的重量, 最后得到的  $n$  个无前缀语言的最轻重量是不会改变的。因此不妨设  $W_a \leq W_b \leq \Lambda \leq W_{ch}$ , 其中  $ch$  表示按字母表顺序的第  $K$  个字母。

设目前找到了  $M$  个单词:  $A_1, A_2, \Lambda, A_{M-1}, A_M$ 。其中任意一个单词都不是其中另一单词的前缀, 且  $W(A_1) \leq W(A_2) \leq \Lambda \leq W(A_{M-1}) \leq W(A_M)$ 。初始时  $M = K, A_1 = a, A_2 = b, \dots, A_M = Ch$ 。

当  $M < N$  时, 还需要加入别的单词, 比如说将单词  $A_1, A_2, \Lambda, A_M$  中的一个去掉, 再添入别的单词 (例如去掉  $A_2$  添入  $A_2a, A_2b, A_2ca, A_2cb\Lambda$ ), 那应该选择哪一个呢? 由于  $A_1, A_2, \Lambda, A_{M-1}, A_M$  不存在前缀关系, 那么如果不考虑重量, 去掉其中任何一个都是等价的。而本题要求语言的总重量最小, 又知道: 去掉  $A_1$  单词后, 目的是最少得到两个单词, 那么重量最少的两个可以同时得到的单词是  $A_1a$  与  $A_1b$ , 也就是说重量至少增加了  $W(A_1a) + W(A_1b) - W(A_1) = W(A_1ab)$ , 那么我们显然要选择第一个单词  $A_1$  (同样可以用反证法证得), 然后  $A_1a, A_1b, \Lambda, A_1Ch$  可以与  $A_2, \Lambda, A_{M-1}, A_M$  合在一起组成  $M + K - 1$  个不存在前缀关系的单词。

当  $M > N$  时, 只要求  $N$  个总重量最小的单词, 由于  $A_1 \leq A_2 \leq \Lambda \leq A_{M-1} \leq A_M$ , 所以  $A_{N+1}, A_{N+2}, \Lambda, A_M$  都可以从语言中去掉。因为用反证法易证: 语言中不可能出现单词:  $A_{N+1}, A_{N+2}, \Lambda, A_M$ 。

当  $M = N$  时, 已经找到了一组解, 但是否这就是最优解呢? 未必吧! (例如: 当  $k = 3, N = 3, W(a) = 1, W(b) = 1, W(c) = 100$  时先找到了 3 个单词  $a, b, c$ , 但  $W(a) + W(b) + W(c) = 100 < W(aa) + W(ab) + W(b) = 5$ ) 从这个例子可以看出: 有可能将某个单词去掉再添入新单词能使得总重量减少。那么去掉哪个单词呢? 这与  $M < N$  时同样的道理: 在不考虑重量的情况下, 去掉  $A_1, A_2, \Lambda, A_{N-1}, A_N$  中任何一个单词是等价的, 而为使语言总重量尽量小, 应该将  $A_1$  去掉添入  $A_1a, A_1b, \Lambda, A_1Ch$ 。

以上过程在什么条件下就可以退出了呢? 设当前最优值为  $Min$  (最初使设为  $\infty$ ), 显然,  $A_1$  的值总是不断扩大的, 所以当  $M = N$  且  $Min \leq A_1N$  则可以退出了。

在上述分析中, 经常用到找一个最小元素, 找一个最大元素 (有时需将最大重量的单词  $A_N$  删掉), 删除某个元素, 插入某个重量的单词等等运算。所以, 用堆做本题是最好不过的, 因为每一种操作的复杂度都是  $O(\log_2 N)$ , 但是, 必须要建一个最大堆和一个最小堆, 并且建立它们之间的映射关系。另外, 由于只要求最小重量, 所以单词是没有必要储存下来的, 只要储存下它的重量就可以了。

源程序:

```
program ex8-11;
const
  Fn1 = 'Naj. In';
  Fn2 = 'Naj. Out';
  MaxN = 10000;
  MaxM = 26;
type
  LinkType = array [1.. MaxN] of Integer;
  HeapType = array [1.. MaxN] of Longint;
```

```

var
  N, M, HeapSize: Integer;
  Min, Now: Longint;
  W: array [1.. MaxM] of Longint;
  MinLink, MaxLink: LinkType;
  MinHeap, MaxHeap: ^HeapType;
procedure Init;
var
  Temp, i, j: Integer;
begin
  Assign (Input, Fnl); Reset (Input);
  Readln (N, M);
  for i := 1 to M do Read (W [i]);
  for i := 1 to M - 1 do
    for j := i + 1 to M do
      if W [i] > W [j] then
        begin
          Temp := W [i]; W [i] := W [j]; W [j] := Temp;
        end;
    Close (Input)
end;
procedure DelMax (A: Longint; L, X: Integer);
var Ch: Integer;
begin
  Ch := X SHL 1;
  while Ch ≤ HeapSize do
    begin
      if (Ch < HeapSize) and (MaxHeap^ [Ch] < MaxHeap^ [Ch + 1]) then Inc (Ch);
      if MaxHeap^ [Ch] ≤ A then Break;
      MaxHeap^ [X] := MaxHeap^ [Ch];
      MaxLink [X] := MaxLink [Ch];
      MinLink [MaxLink [X]] := X;
      X := Ch; Ch := X SHL 1
    end;
  while (X > 1) and (A > MaxHeap^ [X SHR 1]) do
    begin
      MaxHeap^ [X] := MaxHeap^ [X SHR 1];
      MaxLink [X] := MaxLink [X SHR 1];
      MinLink [MaxLink [X]] := X;
    end;
end;

```

```

    X := X SHR 1
  end;
  MaxHeap^ [X] := A; MaxLink [X] := L; MinLink [L] := X
end;
procedure DelMin (A: Longint; L, X: Integer);
var Ch: Integer;
begin
  Ch := X SHL 1;
  while Ch ≤ HeapSize do
    begin
      if (Ch < HeapSize) and (MinHeap^ [Ch] > MinHeap^ [Ch + 1]) then Inc (Ch);
      if MinHeap^ [Ch] ≥ A then Break;
      MinHeap^ [X] := MinHeap^ [Ch];
      MinLink [X] := MinLink [Ch];
      MaxLink [MinLink [X]] := X;
      X := Ch; Ch := X SHL 1
    end;
  while (X > 1) and (A < MinHeap^ [X SHR 1]) do
    begin
      MinHeap^ [X] := MinHeap^ [X SHR 1];
      MinLink [X] := MinLink [X SHR 1];
      MaxLink [MinLink [X]] := X;
      X := X SHR 1
    end;
  MinHeap^ [X] := A; MinLink [X] := L; MaxLink [L] := X
end;

procedure JoinMin;
var
  X: Integer;
  Temp: Longint;
begin
  X := HeapSize;
  while (X > 1) and (MinHeap^ [X] < MinHeap^ [X SHR 1]) do
    begin
      Temp := MinHeap^ [X];
      MinHeap^ [X] := MinHeap^ [X SHR 1];
      MinHeap^ [X SHR 1] := Temp;
      Temp := MinLink [X];

```

```

MinLink [X] := MinLink [X SHR 1];
MinLink [X SHR 1] := Temp;
MaxLink [MinLink [X]] := X;
X := X SHR 1;
MaxLink [MinLink [X]] := X
end
end;

procedure JoinMax;
var
  X: Integer;
  Temp: Longint;
begin
  X := HeapSize;
  while (X > 1) and (MaxHeap^ [X] > MaxHeap^ [X SHR 1]) do
    begin
      Temp := MaxHeap^ [X];
      MaxHeap^ [X] := MaxHeap^ [X SHR 1];
      MaxHeap^ [X SHR 1] := Temp;
      Temp := MaxLink [X];
      MaxLink [X] := MaxLink [X SHR 1];
      MaxLink [X SHR 1] := Temp;
      MinLink [MaxLink [X]] := X;
      X := X SHR 1;
      MinLink [MaxLink [X]] := X
    end
  end;

procedure Join (W: Longint);
begin
  if HeapSize = N then
    begin
      if W ≥ MaxHeap^ [1] then Exit;
      Dec (Now, MaxHeap^ [1]); Dec (HeapSize);
      DelMin (MinHeap^ [N], MinLink [N], MaxLink [1]);
      DelMax (MaxHeap^ [N], MaxLink [N], 1)
    end;
  Inc (Now, W); Inc (HeapSize);
  MinHeap^ [HeapSize] := W; MaxHeap^ [HeapSize] := W;
  MinLink [HeapSize] := HeapSize; MaxLink [HeapSize] := HeapSize;

```

```

JoinMin; JoinMax
end;

procedure GetFirst (var X: Longint);
begin
  X := MinHeap^[1]; Dec (Now, X); Dec (HeapSize);
  DelMax (MaxHeap^[HeapSize + 1], MaxLink [HeapSize + 1], MinLink [1]);
  DelMin (MinHeap^[HeapSize + 1], MinLink [HeapSize + 1], 1)
end;

procedure Calc;
var
  i: Integer;
  X: Longint;
begin
  Min := MaxLongint; Now := 0;
  GetMem (MinHeap, Longint (N) SHL 2);
  GetMem (MaxHeap, Longint (N) SHL 2);
  for i := 1 to M do Join (W [i]);
  while (HeapSize < N) or (Min ≥ MinHeap^[1] * N) do
    begin
      if (HeapSize = N) and (Min > Now) then Min := Now;
      GetFirst (X);
      for i := 1 to M do Join (X + W [i])
    end
  end;
end;

procedure Print;
begin
  Assign (Output, Fn2); Rewrite (Output);
  Writeln (Min);
  Close (Output)
end;

begin
  Init;
  Calc;
  Print
end.

```

## 8.7 小结

在这一章中我们学习多种用于查找的数据结构。这些数据结构基本上是基于集合的存储结构，因此又可以叫做集合抽象数据结构。这些数据结构各有特点，能够反映一个集合中元素的性质，适用于不同的查找需求。顺序结构实现快捷简便，适用于小规模的数据查找，但是由于复杂度太高，大数据的查找就无能为力了。二分查找效率高，容易编写，理论复杂度低，实际效果也非常好，只是需要对输入数据进行有序化，也就是说不是所有的数据都能够用二分查找。索引查找适用建立大规模的数据库，但在信息学中应用不大。哈希表的运用十分灵活，哈希函数设计多样，避免冲突的方法也有很多。哈希表效率很高，有时可以取代树表结构，甚至可以看成  $O(1)$ 。合理巧妙地利用哈希表，在信息学中常常能取到意想不到的效果。树表查找在信息学竞赛有大量的应用，而且树型数据结构有许多种，本章中只列出了其中比较常用的一部分。其他运用比较的数型结构还有红黑树、treap、左高树、树状数组等等。有关树的数据结构问题还有树型题目在最近的比赛中层出不穷，以至成为一种潮流，所以熟练地掌握有关树的知识是十分必要的。

在实际应用查找数据结构的时候，不能总是将所学的经典模型生硬地套在新遇到的题目上。应该具体问题具体分析，找到合适的数据结构后，进行相应的改造和扩充，使其最好地运用于题目中，而不是满足于套上去能用就好。特别是在最近的信息学竞赛中，考的不是单一的某种数据结构，而是多种数据结构的综合运用，因此要求选手不仅要掌握好各种数据结构，还要能够将各种数据结构联合起来，具有创造性的改造。

## 习题八

### 一、单选题

1. 对长度为 10 的顺序表进行查找，若查找前面 5 个元素的概率相同，均为  $1/8$ ，查找后面 5 个元素的概率相同，均为  $3/40$ ，则查找任一元素的平均查找长度为( )。

- A. 5.5                      B. 5                      C.  $39/8$                       D.  $19/4$

2. 对长度为 3 的顺序表进行查找，若查找第一个元素的概率为  $1/2$ ，查找第二个元素的概率为  $1/3$ ，查找第三个元素的概率为  $1/6$ ，则查找任一元素的平均查找长度为( )。

- A.  $5/3$                       B. 2                      C.  $7/3$                       D.  $4/3$

3. 对长度为  $n$  的单链有序表，若查找每个元素的概率相等，则查找任一元素的平均查找长度为( )。

- A.  $n/2$                       B.  $(n+1)/2$                       C.  $(n-1)/2$                       D.  $n/4$

4. 对于长度为 9 的顺序存储的有序表，若采用二分查找，在等概率情况下的平均查找长度为( )的值除以 9。

- A. 20                      B. 18                      C. 25                      D. 22

5. 对于长度为 18 的顺序存储的有序表，若采用二分查找，则查找第 15 个元素的查找长度为( )。

A. 3                      B. 4                      C. 5                      D. 6

6. 在索引查找中, 若用于保存数据元素的主表的长度为  $n$ , 它被均分为  $k$  个子表, 每个子表的长度均为  $n/k$ , 则索引查找的平均查找长度为( )。

A.  $n+k$                       B.  $k+n/k$                       C.  $(k+n/k)/2$                       D.  $(k+n/k)/2+1$

7. 在索引查找中, 若用于保存数据元素的主表的长度为 117, 它被均分为 9 个子表, 则索引查找的平均查找长度为( )。

A. 11                      B. 12                      C. 13                      D. 9

8. 若根据数据集合  $\{23, 44, 36, 48, 52, 73, 64, 58\}$  建立散列表, 采用  $h(K) = K \% 13$  计算散列地址, 并采用链接法处理冲突, 则元素 64 的散列地址为( )。

A. 4                      B. 8                      C. 12                      D. 13

9. 若根据数据集合  $\{23, 44, 36, 48, 52, 73, 64, 58\}$  建立散列表, 采用  $h(K) = K \% 7$  计算散列地址, 则同义词元素的个数最多为( )个。

A. 1                      B. 2                      C. 3                      D. 4

10. 在采用线性探测法处理冲突的散列表上, 假定装填因子  $\alpha$  的值为 0.5, 则查找任一元素的平均查找长度为( )。

A. 1                      B. 1.5                      C. 2                      D. 2.5

11. 在采用链接法处理冲突的散列表上, 假定装填因子  $\alpha$  的值为 4, 则查找任一元素的平均查找长度为( )。

A. 3                      B. 3.5                      C. 4                      D. 2.5

12. 在散列查找中, 平均查找长度主要与( )有关。

A. 散列表长度                      B. 散列元素的个数  
C. 装填因子                      D. 处理冲突方法

13. 在 5 阶 B-树中, 每个非树根结点最多允许有( )个关键字。

A. 2                      B. 3                      C. 4                      D. 5

14. 在一棵含有  $n$  个关键字的 B-树中, 所有结点中的空指针数为( )个。

A.  $n$                       B.  $n+1$                       C.  $n-1$                       D.  $2n$

## 二、填空题

1. 对于二分查找所对应的判定树, 它既是一棵\_\_\_\_\_, 又是一棵\_\_\_\_\_。

2. 假定对长度  $n=50$  的有序表进行二分查找, 则对应的判定树高度为\_\_\_\_\_, 判定树中前 5 层的结点数为\_\_\_\_\_, 最后一层的结点数为\_\_\_\_\_。

3. 假定一个集合为  $\{12, 23, 74, 55, 63, 40, 82, 36\}$ , 若按  $Key \% 3$  条件进行划分, 使得同一余数的元素成为一个子集合, 则得到三个子集合分别为\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_。

4. 在索引表中, 若一个索引项对应主表中的一条记录, 则称此索引为\_\_\_\_\_索引; 若对应主表中的若干条记录, 则称此索引为\_\_\_\_\_索引。

5. 假定对数据集合  $\{38, 25, 74, 52, 48\}$  进行散列存储, 采用  $H(K) = K \% 7$  作为散列函数, 若分别采用线性探查法和链接法处理冲突, 则对各自散列表进行查找的平均查找长度分别为\_\_\_\_\_和\_\_\_\_\_。

6. 假定要对长度  $n=100$  的数据集合进行散列存储, 并采用链接法处理冲突, 则对于长度

$m=20$ 的散列表, 每个散列地址的单链表的长度平均为\_\_\_\_\_。

7. 在数据表的散列存储中, 装填因子  $a$  又称为装填系数, 若用  $m$  表示散列表的长度,  $n$  表示待散列存储的元素个数, 则  $a$  等于\_\_\_\_\_。

8. 在数据表的散列存储中, 处理冲突有\_\_\_\_\_和\_\_\_\_\_两种方法。

9. 对于一棵含有  $N$  个关键字的  $m$  阶 B 树, 其最小高度为\_\_\_\_\_, 最大高度为\_\_\_\_\_。

10. 已知一棵 3 阶 B 树中含有 50 个关键字, 则该树的最小高度为\_\_\_\_\_, 最大高度为\_\_\_\_\_。

11. 在一棵 9 阶的 B 树中, 每个非树根结点的关键字数目最少为\_\_\_\_\_个, 最多为\_\_\_\_\_个。

### 三、运算题

1. 假定查找有序表  $A[25]$  中每一元素的概率相等, 试分别求出进行顺序、二分和分块 (假定被分为 5 块, 每块 5 个元素) 查找每一元素时的平均查找长度。

2. 假定一个待散列存储的数据集合为  $\{32, 75, 29, 63, 48, 94, 25, 46, 18, 70\}$ , 散列地址空间为  $HT[13]$ , 若采用除留余数法构造散列函数并用线性探查法处理冲突, 试求出每一元素的散列地址, 画出最后得到的散列表, 求出平均查找长度。

3. 假定一个待散列存储的数据集合为  $\{32, 75, 29, 63, 48, 94, 25, 36, 18, 70\}$ , 散列地址空间为  $HT[11]$ , 若采用除留余数法构造散列函数并用链接法处理冲突, 试求出每一元素的散列地址, 画出最后得到的散列表, 求出平均查找长度。

4. 已知一组关键字为  $\{26, 38, 12, 45, 73, 64, 30, 56\}$ , 试依次插入关键字生成一棵 3 阶的 B 树, 画出每次插入一个关键字后 B 树的结构。

5. 已知一棵 4 阶 B 树如图 8-22 所示, 假定依次从中删除关键字 46, 24, 52, 8, 93, 80, 试画出每删除一个关键字后 B 树的结构。

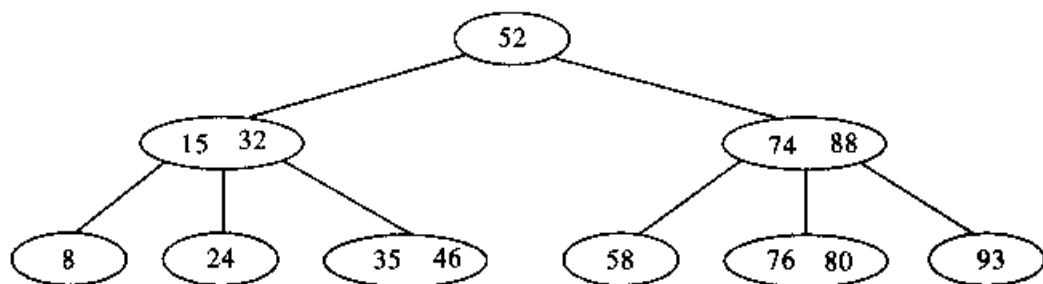


图 8-22 4 阶 B 树

### 四、上机编程题

1. 编写一个非递归算法, 在稀疏有序索引表中二分查找出给定值  $K$  所对应的索引项, 即索引值刚好大于等于  $K$  的索引项, 返回该索引项的  $start$  域的值, 若查找失败则返回 -1。

2. 假定有一个  $100 \times 100$  的稀疏矩阵, 其中 1% 的元素为非零元素, 现要求对其非零元素进行散列存储, 使之能够按照元素的行、列值存取矩阵元素 (即元素的行、列值联合为元素的关键字), 试采用除留余数法构造散列函数并用线性探查法处理冲突, 分别写出建立散列表和查找散列表的算法。



3. 给定一个正整数序列  $A[1..n]$ , 要求你设计一个算法, 使之能在尽量快的时间内完成一下两种操作:

- a) 给定正整数  $i, p$ , 即将  $a[i]$  更改为  $p$ ;
- b) 给定  $l, r, k$ , 即, 询问区间  $[l..r]$  中第  $k$  大的数字。

4. 给定二维坐标系中的  $n$  个点  $V = (X[1..n], Y[1..n])$  和  $m$  次如下形式的询问:

对于  $(x_0, y_0)$  与  $(x_1, y_1)$ , 满足  $x_0 \leq x_1$  与  $y_0 \leq y_1$ , 求有多少个点满足  $x_0 \leq x[i] \leq x_1$  且  $y_0 \leq y[i] \leq y_1$ 。

这里我们规定:  $n \leq 10000$ ;  $m \leq 100000$ 。

5. 鹰蛋:

有一堆共  $M$  个鹰蛋, 一位教授想研究这些鹰蛋的坚硬度  $E$ 。他是通过不断从一幢  $N$  层的楼上向下扔鹰蛋来确定  $E$  的。当鹰蛋从第  $E$  层楼及以下楼层落下时是不会碎的, 但从第  $(E+1)$  层楼及以上楼层向下落时会摔碎。如果鹰蛋未摔碎, 还可以继续使用; 但如果鹰蛋全碎了却仍未确定  $E$ , 这显然是一个失败的实验。教授希望实验是成功的。

例如: 若鹰蛋从第 1 层楼落下即摔碎,  $E=0$ ; 若鹰蛋从第  $N$  层楼落下仍未碎,  $E=N$ 。

这里假设所有的鹰蛋都具有相同的坚硬度。给定鹰蛋个数  $M$  ( $M \leq 100000$ ) 与楼层数  $N$  ( $N \leq 1000000000$ )。要求最坏情况下确定  $E$  所需要的最少次数。

样例:

$M=1, N=10$

$ANS=10$

样例解释: 为了不使实验失败, 只能将这个鹰蛋按照从一楼到十楼的顺序依次扔下。一旦在第  $(E+1)$  层楼摔碎,  $E$  便确定了。(假设在第  $(N+1)$  层摔鹰蛋会碎)

## 9 排序

### 9.1 排序的基本概念

排序是一种很基本的算法,有的题目关键问题决定于排序。排序的方法多种多样,但它们的时空复杂度又各不相同,怎样在空间复杂度允许的前提下,降低它的时间复杂度,是竞赛过程中程序能否通过大数据量的测试的关键,所以必须掌握各种排序的方法,便于在竞赛时使用。本章对几种常见的排序方法进行一些分析和比较,以利于全面掌握排序算法。

在一般情况下,排序问题的输入是  $n$  个数  $a_1, a_2, a_3, \dots, a_n$  的一个序列,要设计一个有效的排序算法,产生输入序列的一个重排  $a'_1, a'_2, a'_3, \dots, a'_n$ , 使得:  $a'_1 \leq a'_2 \leq a'_3 \leq \dots \leq a'_n$ 。输入序列通常是一个有  $n$  个元素的数组。当然也可以用其他形式来表示。在实际中,待排序的对象往往不是单一的数,而是一个记录,其中有一个关键字域  $key$ ,它是排序的根据。在  $key$  的数据类型上定义了某个线性序列。例如,整数、实数、字符串等都可以作为关键字。记录的其他数据称为卫星数据,即它们都是以  $key$  为中心的。在一个实际的排序算法中,当对关键字重排时,卫星数据也会跟着关键字一起移动。如果每个记录都很大,可以对一组分别指向各个不同记录的指针进行排序,以求减少数据的移动量。对于排序算法来说,不论待排序对象是单个数值,还是记录,它们的排序方法都是一样的。在排序时,待排序记录的关键字可能有相同者,对于关键字相同的记录通常并不要求它们之间应怎样排列,只要求在最后输出时,关键字小者排在关键字大者之前。

记录的关键字可以是记录的关键字或非关键字,所以关键字相同的记录可能只有一个,也可能有多个。对于具有同一关键字的多个记录来说,若采用的排序方法使排序后记录的相对次序不变,则称此排序方法是稳定的,否则称为不稳定的。例如,有一组记录的关键字为 (23, 85, 72, 58, 23, 40), 其中关键字同为 23 的记录有两个(为了区分,后一个记录关键字 23 下带有下划线)。若一种排序方法使排序后的结果为 (23, 23, 40, 58, 72, 85), 则称此方法是稳定的;若一种排序方法使排序后的结果为 (23, 23, 40, 58, 72, 85), 则称此方法是不稳定的。

对排序计算时间的分析可以遵循若干种不同的准则,通常以排序过程所需要的算法步数作为度量,有时也以排序过程中所作的关键字的比较次数作为度量。特别是当关键字比较需要较长时间时,例如,当关键字是较长字符串时,常以关键字比较次数作为排序算法计算时间复杂度的度量。当排序时需要移动记录,且记录又很大时,还应该考虑记录的移动次数。究竟采用哪种度量方法比较合适要根据具体情况而定。

## 9.2 简单排序算法

### 一、冒泡排序

最简单的排序方法是冒泡排序方法。这种排序方法的基本思想是：将待排序的记录看作是竖着排列的“气泡”，关键字较小的记录比较轻，从而要往上浮。在冒泡排序算法中我们要对这个“气泡”序列处理若干遍。所谓一遍处理，就是自底向上检查一遍这个序列，并时刻注意两个相邻的记录的正确顺序。如果发现两个相邻记录的顺序不对，即“轻”的记录在下面，就交换它们的位置。显然，处理一遍之后，“最轻”的记录就浮到了最高位置，处理两遍之后，“次轻”的记录就浮到了次高位置。在作第二遍处理时，由于最高位置上的记录已是“最轻”记录，因为经过前面  $i-1$  遍的处理，它们已正确地排好序。

7	5	3	8	6	8 和 6 比较, $8 > 6$ , 6 上浮, 即 8 和 6 进行对调
7	5	3	6	8	6 和 3 比较, $6 > 3$ , 3 已在上面, 即 6 和 3 不进行对调
7	5	3	6	8	3 和 5 比较, $3 < 5$ , 3 上浮, 即 3 和 5 进行对调
7	3	5	6	8	3 和 7 比较, $3 < 7$ , 3 上浮, 即 3 和 7 进行对调
3	7	5	6	8	

图 9-1 一遍处理的检查上浮图

从图 9-1 可以看出，按上述排序思想的算法，对  $N$  个记录的排序，进行一遍处理后，“最轻”的记录逐步上浮到第一个位置上，然后进行第二遍处理，“次轻”的记录逐步上浮到第二个位置上……经过  $N-1$  次这样的处理后，所有的记录就都正确到位了，从而实现了排序。这个算法可实现如下：

```
for i: = 1 to n-1 do
```

```
for j: = n downto i+1 do
```

```
if A[j].key < A[j-1].key then swap (A[j], A[j-1])
```

其中  $A$  是一个记录数组， $\text{swap}(A[j], A[j-1])$  是一个对调  $A[j]$  和  $A[j-1]$  的过程。

**例题 9-1** 从键盘上输入十个正整数，把这十个数按从小到大的顺序排列。

分析：

本题用冒泡法实现，算法分析与说明在上面已详细介绍，这里加入一个标志变量  $\text{flag}$ ，用来表示每一趟排序是否有交换的标志，在每趟进行之前置为 0，进行一趟后若无交换，则表明已是有序数列，排序过程结束；否则，表明数据列还没有完全有序，需要继续排序过程。这样改进后冒泡排序的效果会好些。这里给出源程序：

```
Program ex9-1;
```

```
Const n = 10;
```

```

Var
  A: array [1..n] of integer;
  i, j, k, flag: integer;
BEGIN
  Writeln ( 'Please input ', n, 'integer number;');
  For i: =1 to n do
    Read (a [i]);
  Readln;
  i: =1;
  Repeat
    Flag: =0;
    For j: =n downto i+1 do
      If a [j] < a [j-1] then
        Begin k: = a [j]; a [j]: =a [j-1]; a [j-1]: =k; flag: =flag+1 end;
    i: =i+1
  Until (i>=n) or (flag=0)
  Writeln ( ' The new square is:');
  For i: =1 to n do
    Begin
      Write (a [i]: 8);
      If i mod 5 =0 then writeln
    End;
END.

```

## 二、插入排序

插入排序的基本思想是：经过  $i-1$  遍处理后， $A[1], A[2], \dots, A[i-1]$  已排好序。第  $i$  遍处理仅将  $A[i]$  插入  $A[1], A[2], \dots, A[i-1]$  的适当位置，使得  $A[1], A[2], \dots, A[i]$  还是排好序的序列。要达到这个目的，可以用从前往后和从后往前依次比较的两种方法。这里采用从后往前依次比较的方法。首先比较  $A[i].key$  和  $A[i-1].key$  比，如果  $A[i-1].key \leq A[i].key$ ，则  $A[1], A[2], \dots, A[i]$  已排好序，第  $i$  遍处理就结束了；否则交换  $A[i-1]$  与  $A[i]$  的位置，继续比较  $A[i-1].key$  和  $A[i-2].key$ ，直到找到某一个位置  $j$  ( $1 \leq j \leq i-1$ )，使得  $A[j].key \leq A[j+1].key$  时为止。为了编程方便，我们引入一个“哨兵”元素  $A[0]$ ，它的关键字小于  $A[1], A[2], \dots, A[n]$  中的任一元素的关键字。下面通过一个具体实例描述插入排序算法的实现过程，如图 9-2 所示。插入排序算法实现过程如下：

i = 1	[8]	3	2	5	9	1	6
i = 2	[3	8]	2	5	9	1	6
i = 3	[2	3	8]	5	9	1	6
i = 4	[2	3	5	8]	9	1	6
i = 5	[2	3	5	8	9]	1	6
i = 6	[1	2	3	5	8	9]	6
i = 7	[1	2	3	5	6	8	9]

图 9-2 插入排序具体示例

```

CONST max = 20;
TYPE
Node = RECORD
    Key: integer;
    Data: integer;
END;
Sarray = array [0..max] of node;
Procedure insert (VAR A: sarray; n: integer);
Var i, j: integer;
BEGIN
    For i := 2 to n do
        Begin
            A [0] := A [i];
            j := i - 1;
            While A [0].key < A [j].key do
                Begin
                    A [j + 1] := A [j];
                    j := j - 1;
                End;
            A [j + 1] := A [0];
        End
    END;

```

### 三、选择排序

选择排序的基本思想是：对待排序的记录序列进行  $n-1$  遍的处理，第  $i$  遍处理是将  $A[i]$ ,  $A[i+1]$ , ...,  $A[n]$  中具有最小关键字者与  $A[i]$  交换位置。这样，经过  $i$  遍处理后，前  $i$  个记录的位置已经是正确的了。下面通过一个具体实例理解选择排序的全过程，如图 9-3 所示。

初始状态:	8	3	2	5	9	1	6
i = 1	[1]	3	2	5	9	6	6
i = 2	[1	2]	3	5	9	8	6
i = 3	[1	2	3]	5	9	8	6
i = 4	[1	2	3	5]	9	8	6
i = 5	[1	2	3	5	6]	8	9
i = 6	[1	2	3	5	6	8]	9
i = 7	[1	2	3	5	6	8	9]

图 9-3 选择排序具体示例

选择排序的算法实现如下:

```

Var
    Lowkey: keytype; {最小关键字}
    Lowindex: integer; {最小关键字所在记录的位置}
Begin
    For i: = 1 to n-1 do
        Begin
            Lowindex: = i;
            Lowkey: = a [ i ] . key;
            For j: = i+1 to n do
                If a [ j ] . key < lowkey then
                    Begin
                        Lowkey: = a [ j ] . key;
                        Lowindex: = j;
                    End;
            Swap ( a [ i ], a [ lowindex ] );
        End
    End;
End;
```

### 四、简单排序算法时间复杂度分析

前面介绍的三种排序算法的时间复杂度都是  $O(n^2)$ 。

首先考虑冒泡排序算法。若在其过程中加入一个标志变量 flag, 用来表示每一趟排序是否有交换, 在每趟进行之前置为 0, 进行一趟后若无交换, 则表明已有序, 排序过程结束。这样改进后冒泡排序的效果会好一些。改进后分析冒泡排序的平均时间复杂度, 最好的情况是输入序列已经有序, 总的比较次数为  $(n-1)$  次, 且不移动元素; 最坏情况是输入序列为逆序, 则需进行  $(n-1)$  趟排序, 其比较次数为  $\sum_{i=1}^{n-1} (n-i) = (n^2 - n) / 2$  次, 移动次数为  $3(n^2 - n) / 2$ ; 在平均情况

下,比较和移动元素的总次数大约为最坏情况下的一半。因此冒泡排序的时间复杂度为  $O(n^2)$ 。由于冒泡排序通常比插入排序和选择排序需要移动元素的次数多,所以它是三种简单排序中速度最慢的一种。

其次考虑插入排序。根据算法可知,为了正确地插入第  $i$  个元素,最少比较一次,最多比较  $(i-1)$  次,平均比较次数为  $i/2$  次,同时移动次数也接近比较次数,因此插入排序平均所需的比较次数和移动次数都是  $(n^2 + n - 2)/4 \approx n^2/4$ , 因此插入排序的时间复杂度也是  $O(n^2)$ 。

最后考虑选择排序。根据算法可知,选择排序的比较次数与初始排列是无关的,第一趟要  $(n-1)$  次比较,第二趟要  $(n-2)$  次比较,依此类推,故总的比较次数为  $n(n-1)/2$ ; 元素的移动需要一个暂存空间,因此移动次数为  $3(n-1)$  次,显然它的时间复杂度为  $O(n^2)$ 。

### 9.3 快速排序

前一节中介绍的几个简单排序算法在最坏情况下都需要  $O(n^2)$  计算时间,有时由于数据量很大,这样一个时间复杂度很可能在规定的时间内出不了解,自然就要优化排序算法,降低排序的时间复杂度。下面研究的快速排序算法就是追求这个目标,它在平均情况下的时间复杂度为  $O(n \log_2 n)$ 。

#### 一、快速排序算法的基本思想及实现

快速排序的基本思想是基于分治策略,是对冒泡排序的一种改进。在冒泡排序中,进行元素(记录)的比较和交换是在相邻单元中进行的,元素(记录)每次交换只能上移或下移一个单元,因而总的比较和移动次数较多;在快速排序中,元素(记录)的比较和交换是从两端向中间进行的,关键字较大的元素(记录)一次就能够交换到后面单元,关键字较小的记录一次就能够交换到前面的单元,记录每次移动的距离较远,因而总的比较和移动次数较少。

快速排序的基本思想是:把  $n$  个记录组成的文件顺序读入内存并用一个数组保存,文件中的每一个记录就是数组中的一个元素。先取数组中某一个元素(通常为第一个)的关键字为控制关键字,相应的数组元素为基准元素。设法把该基准元素放到数组中合适的位置上,同时对其他数组元素做适当调整,使得在这个基准元素的右面的所有数组元素的关键字均大于基准元素的关键字,而在它左面的那些数组元素的关键字小于基准元素的关键字,基准元素的当前位置就是排序后的最终位置;然后再对基准元素的前后两个子区间分别进行快速排序(即重复上述过程),直到每个区间为空或只包含一个元素时,整个快速排序结束。

在快速排序中,把待排序的区间按照第一个元素(即基准元素)的关键字分为前后(或称左右)两个子区间的过程称为一次划分。实现一次划分过程是:设有  $n$  个元素的关键字用  $k_1 \sim k_n$  来表示,设立两个位置指针  $i$  和  $j$ 。初始时,令  $i=1, j=n$ ; 比较  $k_i$  和  $k_j$ , 如果  $k_i \leq k_j$ , 则修改  $j$  ( $j:=j-1$ ); 再将  $i, j$  位置上的关键字进行比较,当  $i < j$  且  $k_i > k_j$  时,将两个元素进行交换;交换位置后,修改  $i$  ( $i:=i+1$ )。重复上述操作,直到  $i=j$ , 则  $i$  所指示的位置就是基准元素的位置。在实际编程中,在未确定基准元素的位置前,并不需要真正的交换存储位置,而可以开设一个暂存单元  $x$  来存放基准元素。待确定位置后,再将其存入。如图 9-4 仅列出了关键字,所以“移动 17”就应理解成移动关键字为 17 的元素,元素移动后的位置用图中  $[]$  表示。另外,“46 送  $x$ ”是指

序号	关键字	说 明
	46    55    13    42    94    05    17    70 ↑         ↑ i         j	46 送 x: 70>46 修改j
(2)	[ ]    55    13    42    94    05    17    [70] ↑         ↑                 ↑ i         j                 j	17<46, 移动 17, 修改 i
(3)	[17]    55    13    42    94    05    [ ]    [70] ↑                             ↑         ↑ i                             i         j	55>46, 移动 55, 修改 j
(4)	[17]    [ ]    13    42    94    05    [55    70] ↑         ↑                 ↑ i         j                 j	05<46, 移动 17, 修改 i
(5)	[17    05]    13    42    94    [ ]    [55    70] ↑         ↑                 ↑ i         j                 j	13<46, 修改 i
(6)	[17    05    13]    42    94    [ ]    [55    70] ↑         ↑                 ↑ i         j                 j	42<46, 修改 i
(7)	[17    05    13    42]    94    [ ]    [55    70] ↑         ↑         ↑ i         j         j	94>46, 移动 94, 修改 j
(8)	[17    05    13    42]    [ ]    [94    55    70] ↑         ↑ i         j	i=j x 送 i 位置
(9)	[17    05    13    42]    46    [94    55    70]	确定 46 的位置

图 9-4 在快速排序中一次划分的过程示例

在该示例过程中，遵循下列原则进行操作：

(1) 在每次比较时,  $i$  或  $j$  是取数指针。比较总是在取数指针所指元素的关键字与基准元素  $x$  的关键字  $k_x$  进行比较。比较后有如下几种情况:

●当大于 46 时:

若取数指针是  $i$ ，则将  $i$  指示的元素送入  $j$  所指示的位置，并修改  $j$  ( $j := j - 1$ )；

若取数指针是  $j$ ，则不移动元素仅修改  $j$  ( $j := j - 1$ )。

●当小于等于 46 时:

若取数指针是  $j$ ，则将  $j$  指示的元素送入  $i$  所指示的位置，并修改  $i$  ( $i := i + 1$ )；

若取数指针是  $i$ ，则不移动元素仅修改  $i$  ( $i := i + 1$ )。

(2) 第一次比较时, 取数指针是  $j$ 。以后, 凡是本次比较后得到修改的指针  $i$  或  $j$ , 则是下一次比较的取数指针。

每次比较后，元素不一定被移动，但指针有且仅有一个得到修改。在找到第一个元素的最终位置后，就把数组分成左右两个部分。在这两个部分中分别重复上述操作，直到每个区间为空或只有一个元素时，快速排序结束。下面给出快速排序的过程：

```
CONST max = 30;
```

**TYPE**

Node = RECORD

**Key:** integer;

Data: integer;





```

END;
Sarray = array [0..max] of node;
Procedure quicksort (VAR A: sarray; s, t: integer);
    Var i, j: integer;
        x: node;
BEGIN
    i := s; j := t; x := A[s]
    while i < j do
        Begin
            while (A[j].key ≥ x.key) and (j > i) do j := j - 1;
            if j > i then begin
                A[i] := A[j];
                i := i + 1
            end;
            while (A[i].key ≤ x.key) and (i < j) do i := i + 1;
            if i < j then begin
                A[j] := A[i]; j := j - 1;
            end;
            A[i] := x
        end;
    End;
    If s < (i - 1) then quicksort (a, s, i - 1);
    If (i + 1) < t then quicksort (a, i + 1, t);
END;

```

## 二、快速排序时间复杂度分析

快速排序附加的存储空间主要是一个栈和一个暂存元素的单元 X。栈中存放排序子文件的首尾位置。如果在入栈时，都选较长的一个子文件，则第一次入栈的子文件长度大于或等于  $n/2$ ；第二次入栈的子文件的长度大于或等于第一次划分后较短子文件长度的一半。依此类推，栈中最多进入  $\log_2 n$  个子文件。所以栈的大小只需能存放  $2\log_2 n$  个整数的空间。因此，快速排序要求附加的存储单元的数量可记为  $O(\log_2 n)$ 。

快速排序的时间复杂度，最坏的情况是待排序的元素已经有序，此时时间为最长。这时，第一趟排序经过  $n-1$  次比较后，将第一个元素仍定在它原来的位置上，并得到一个有  $n-1$  个元素的子文件；第二趟排序，经过  $n-2$  次比较，将第二个记录仍定在它原来的位置上，并得到一个包含  $n-2$  个元素的子文件；依此类推，所以，总比较次数为：

$$(n-1) + (n-2) + \cdots + 1 = n(n-1)/2, \text{ 记为 } O(n^2)。$$

另一种特别情况是最好情况，即每趟比较后，基准元素的位置正好在文件的中央，从而把文件分成大小相等的两个子文件，其总的比较次数为：

$$T(n) \leq n + 2T(n/2) \leq 2n + 4T(n/4) \leq 3n + 8T(n/8) \cdots \leq n\log_2 n + NT(1)$$

所以总的比较次数为  $O(n \log_2 n)$ 。可以证明, 平均比较次数也是  $O(n \log_2 n)$ 。

### 三、随机快速排序

通过上面的分析知道, 快速排序的性能取决于划分的对称性, 通过修改对数组进行划分的基准元素, 可以设计出采用随机选择策略的快速排序算法。在快速排序算法的每趟中, 当数组还没有划分时, 可以在数组中随机选出一个元素作为划分的基准元素, 这样可以使划分基准的选择是随机的, 从而可以期望划分是较对称的。

随机化快速排序过程只需修改每次划分的基准元素就可以了, 对于上面快速排序的算法中, 设  $t$  是待排序区间的最后一个元素的下标,  $s$  是待排序区间的最前一个元素的下标, 则只需修改  $k$ :  $k = \text{random}(t - s + 1)$ ; 令基准元素  $x_k = A[k]$  即可。

## 9.4 堆排序

堆排序是利用堆的特性进行排序的过程。下面首先给出堆的定义: 假设有一个元素序列为  $\{R_1, R_2, \dots, R_n\}$ , 对应的关键字序列为  $\{S_1, S_2, \dots, S_n\}$ , 若此关键字序列满足下列任一种特性则称此元素序列 (或以关键字序列代之) 为堆。

(1)  $S_i \leq S_{2i}$  和  $S_i \leq S_{2i+1}$  ( $1 \leq i \leq n/2$ )

(2)  $S_i \geq S_{2i}$  和  $S_i \geq S_{2i+1}$  ( $1 \leq i \leq n/2$ )

若满足第一种特性则称此堆为小根堆, 若满足第二种特性则称此堆为大根堆。这里只讨论大根堆, 其中的规则只需稍加修改即适合小根堆。

一个堆对应一棵完全二叉树, 树中每个编号为  $i$  的结点的值就是堆中下标为  $i$  的元素  $R_i$ , 图 9-5 (a) 所示就是一个具有 10 个元素的堆所对应的完全二叉树。根据堆的定义, 若一棵完全二叉树是堆, 则该树中每个非终端结点的关键字必然大于等于它的左、右孩子关键字, 若不满足这个条件, 则不是一个堆。例如关键字序列 (75, 38, 62, 25, 16, 49) 就是一个堆 (即大根堆), 其对应的完全二叉树和顺序存储结构如图 9-5 (b) 和 (c) 所示。

堆排序包括构成初始堆和利用堆排序两个阶段。

构建初始堆就是把待排序的元素序列  $\{R_1, R_2, \dots, R_n\}$ , 按照堆的定义调整为堆  $\{R_1', R_2', \dots, R_n'\}$ , 其中  $S_i' \geq S_{2i}'$  和  $S_i' \geq S_{2i+1}'$ ,  $1 \leq i \leq n/2$ 。为此需从对应完全二叉树中编号最大的分支结点 (即编号为  $n/2$  的结点) 起, 至整个树根结点 (即编号为 1 的结点) 止, 依次对每个分支结点进行“筛”运算, 以便形成以每个分支结点为根的堆, 当最后对树根结点进行筛运算后, 整个树就构成一个堆。

下面讨论如何对每个分支结点  $R_i$  ( $1 \leq i \leq n/2$ ) 进行筛运算, 以便构成以  $R_i$  为根的堆。因为, 当对  $R_i$  进行筛运算时, 比它编号大的分支结点都已进行过筛运算, 即已形成了以各个分支结点为根的堆, 其中包括以  $R_i$  的左、右孩子结点  $R_{2i}$  和  $R_{2i+1}$  为根的堆 (若  $R_{2i}$  和  $R_{2i+1}$  为叶子结点, 则认为叶子结点自然为堆), 所以, 对  $R_i$  进行筛运算是在其左、右子树均为堆的基础上实现的。筛运算的过程可描述为: 首先把  $R_i$  的关键字  $S_i$  与两个孩子中关键字较大者  $S_j$  ( $j = 2i$  或  $2i + 1$ ) 进行比较, 若  $S_i \geq S_j$ , 则以  $S_i$  为根的子树成为堆, 筛运算完毕; 否则,  $R_i$  与  $R_j$  互换位置, 互换后可能破坏以  $R_j$  (此时的  $R_j$  的值为

原来的  $R_i$  为根的堆,接着再把  $R_i$  与它的两个孩子中关键字较大者进行比较,依此类推,直到父结点的关键字大于等于孩子结点中较大的关键字或者孩子结点为空时止。这样,以  $R_i$  为根的子树就被调整为一个堆。在对  $R_i$  进行筛运算中,若它的关键字较小,则会被逐层下移,就像过筛子一样,小的被漏下去,大的被选上来,所以把构成堆的过程形象地称为筛运算。

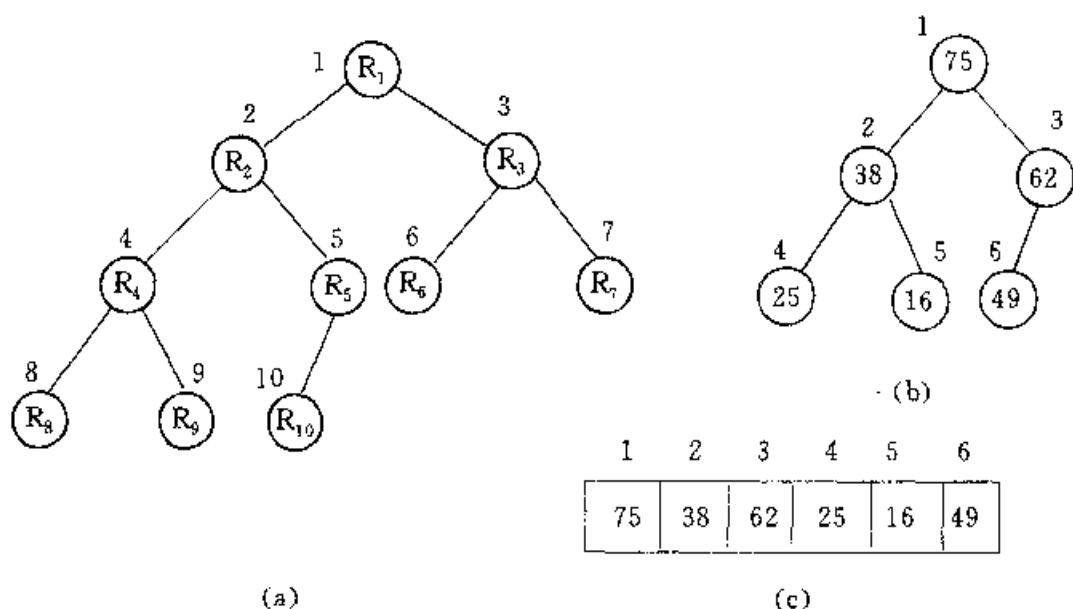


图 9-5 堆的完全二叉树表示

图 9-6 给出了对待排序元素的关键字序列 (45, 36, 18, 53, 72, 30, 48, 93, 15, 36) 构成初始堆的全过程。因结点数  $n=10$ , 所以从第 5 个结点起至第一个结点止, 依次对每个结点进行筛运算。图 9-6 (a) 为按照原始关键字序列所构成的完全二叉树, 图 9-6 (b) ~ (f) 为依次对每个分支结点进行筛运算后所得到的结果, 其中 (f) 为最后构成的初始堆。

假定待排序的  $n$  个元素存放于一维数组  $A[1..N]$  中, 则对  $A[i]$  进行筛运算的算法描述为:

Procedure heap ( $A, n, I$ );

Begin

$X := A[I];$  {把待筛结点的值存于辅助变量  $X$ }

$J := 2 * I;$  { $A[J]$  是  $A[I]$  的左孩子}

While  $j \leq n$  do

begin

If  $(j < n)$  and  $(a[j].key < a[j+1].key)$  do  $j := j+1;$

{若右孩子的关键字较大, 则把  $J$  修改为右孩子的下标}

If  $x.key < a[j].key$  then

begin

$A[I] := a[j];$  {将  $A[J]$  调到双亲位置上}

$I := j; j := 2 * I;$  {修改  $I$  和  $J$  的值, 以便继续向下筛}

end

Else  $j := n+1;$  {筛运算完成, 令  $J = N+1$ , 以便中止循环}

End;

$A[1] := x;$  {被筛结点的值放入最终位置}

End;

根据堆的定义和上面建堆的过程可以知道,编号为1的结点 $A[1]$ (即堆顶)是堆中 $n$ 个结点中关键字最大的结点。所以利用堆排序的过程比较简单,首先把 $A[1]$ 与 $A[n]$ 对换,使 $A[n]$ 为关键字最大的结点,接着对 $A[1]$ (即对调前的 $A[n]$ )在前 $(n-1)$ 个结点中进行筛运算,又得到 $A[1]$ 为当前区间内具有最大关键字的结点,再接着把 $A[1]$ 同当前区间的最后一个结点 $A[n-1]$ 对换,使 $A[n-1]$ 为次最大关键字结点,这样经过 $(n-1)$ 次对换和筛运算后,所有结点成为有序,排序结束。

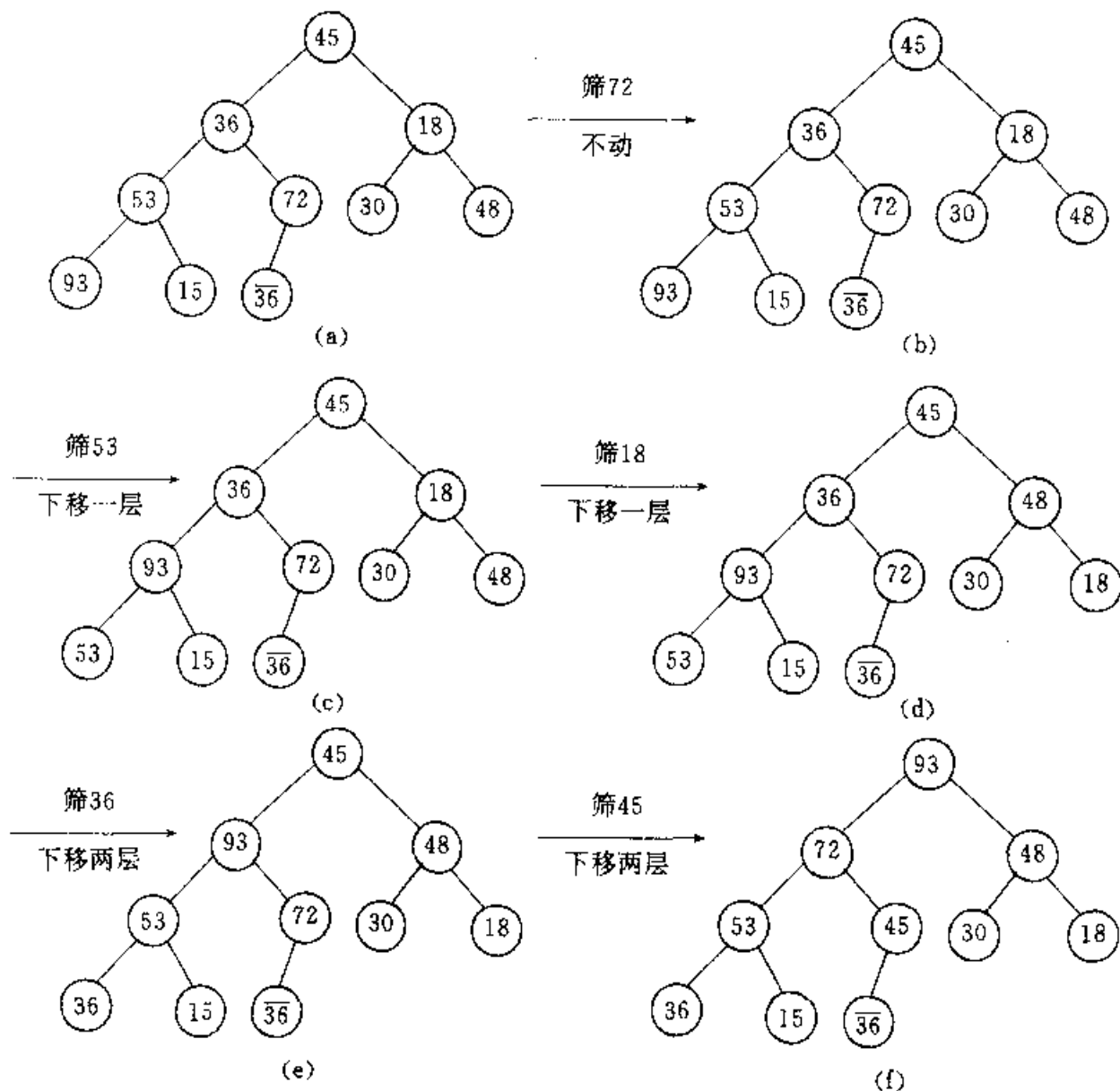


图 9-6 构成初始堆的图形示例

假定在图 9-6 (f) 已构成堆的基础上进行堆排序,则前 4 次对换和筛运算的过程如图 9-7

所示。堆排序的算法可描述为：

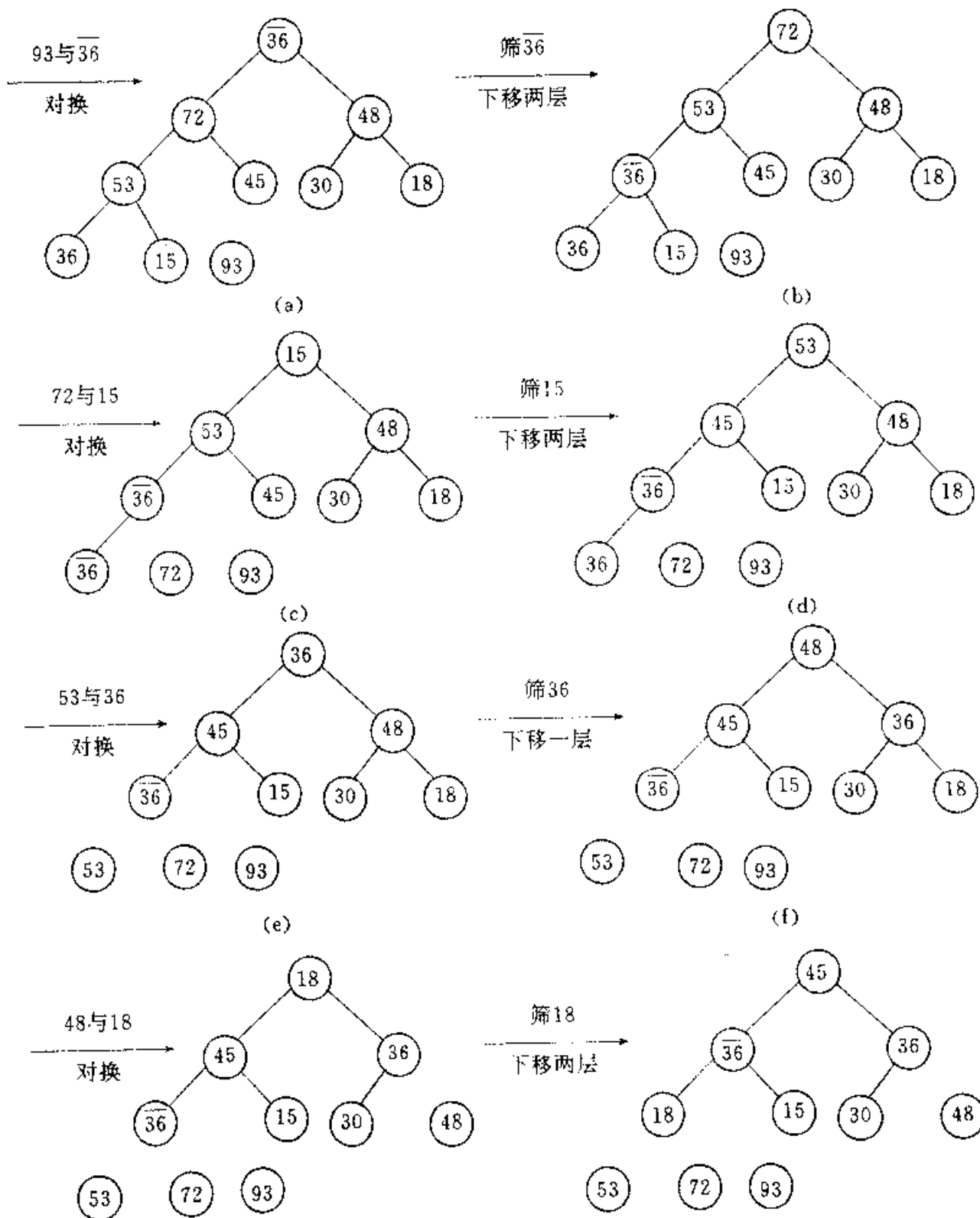


图 9-7 堆排序的图形示例

Procedure heapsort (A, n);

Begin

For  $i := (n \text{ div } 2) \text{ downto } 1$  do

    Heap (A, n, i); {建立初始堆}

For  $i := n \text{ downto } 2$  do {进行  $N-1$  次循环, 完成堆排序}

begin

    SWAP (A [1], A [i]); {将第一个元素同当前区间内最后一个元素对换}

    Heap (A, i-1, 1); {筛 A [1] 结点, 得到 (i-1) 个结点的堆}

End;

假定  $n=8$ , 数组 A 中 8 个元素的关键字为 (36, 25, 48, 12, 65, 43, 20, 58), 图 9-8 (a) 和 (b) 分别给出了在构成初始堆和利用堆排序的过程, 每次筛运算后数组 A 中各元素关键字的变动的情况。

下标	1	2	3	4	5	6	7	8
(0)	36	25	48	12	65	43	20	58
(1)	36	25	48	58	65	43	20	12
(2)	36	25	48	58	65	43	20	12
(3)	36	65	48	58	25	43	20	12
(4)	65	58	48	36	25	43	20	12

(a) 构成初始堆的过程

下标	1	2	3	4	5	6	7	8
(0)	65	58	48	36	25	43	20	12
(1)	58	36	48	12	25	43	20	65
(2)	48	36	43	12	25	20	58	65
(3)	43	36	20	12	25	48	58	65
(4)	36	25	20	12	43	48	58	65
(5)	25	12	20	36	43	48	58	65
(6)	20	12	25	36	43	48	58	65
(7)	12	20	25	36	43	48	58	65

(b) 利用堆排序的过程

图 9-8 堆排序的全过程

在整个堆排序中, 共需要进行约  $3n/2$  次筛运算, 每次筛运算进行父子或兄弟结点的关键字的比较次数和元素的移动次数都不会超过完全二叉树的高度, 所以筛运算的时间复杂度为  $O(\log_2 n)$ , 故整个堆排序过程的时间复杂度为  $O(n \log_2 n)$ 。另外, 由于在堆排序过程中需要进行不相邻位置间元素的移动和交换, 所以它不是一种不稳定的排序方法。

## 9.5 归并排序

在讨论归并排序之前, 首先给出归并的概念。归并 (merge) 就是将两个或多个有序表合并成一个有序表。若将两个有序表合并成一个有序表则称为二路归并, 同理, 有三路归并、四路归并等。二路归并最为简单, 且适应于内排序, 所以我们只讨论二路归并。例如有两个有序表 (7, 10, 13, 15) 和 (4, 8, 19, 20), 归并后得到的有序表为 (4, 7, 8, 10, 13, 15, 19, 20)。

二路归并算法很简单, 假定待归并的两个有序表分别存于数组 A 中从下标 s 到下标 m 的单元和从下标 m+1 到下标 t 的单元 ( $s \leq m, m+1 \leq t$ ), 结果有序表存于数组 R 中从下标 s 到下标 t 的单元, 并令 i, j, k 分别指向这些有序表的第一个单元。归并过程为: 比较 A [i] . stn 和

$A[j].\text{stn}$  的大小, 若  $A[i].\text{stn} \leq A[j].\text{stn}$ , 则将第一个有序表中的元素  $A[i]$  复制到  $R[k]$  中, 并令  $i$  和  $k$  分别加 1, 即使之分别指向后一单元, 否则将第二个有序表中的元素  $A[j]$  复制到  $R[k]$  中, 并令  $j$  和  $k$  分别加 1; 如此循环下去, 直到其中的一个有序表比较和复制完, 然后再将另一个有序表中剩余的元素复制到  $R$  中从下标  $k$  到下标  $t$  的单元。

归并排序 (Merge Sorting) 就是利用归并操作把一个无序表排列成一个有序表的过程。若利用二路归并操作则称为二路归并排序。二路归并排序的过程是: 首先把待排序区间 (即无序表) 中的每一个元素都看作为一个有序表, 则  $n$  个元素构成  $n$  个有序表, 接着两两归并 (即第一个表同第二个表归并, 第三个表同第四个表归并, 等等, 若最后只剩下一个表, 则直接进入下一趟归并), 得到  $\lfloor n/2 \rfloor$  个长度为 2 (最后一个表的长度可能小于 2) 的有序表, 称此为一趟归并, 然后再两两有序表归并, 得到  $\lfloor \lfloor n/2 \rfloor / 2 \rfloor$  个长度为 4 (最后一个表的长度可能小于 4) 的有序表, 如此进行下去, 直到归并第  $\lceil \log_2 n \rceil$  趟后得到一个长度为  $n$  的有序表为止。

例如, 有 12 个元素的排序码为:

(45, 36, 18, 53, 72, 30, 48, 93, 15, 24, 65, 47)

则进行二路归并排序的过程如图 9-9 所示:

(0) {45} {36} {18} {53} {72} {30} {48} {93} {15} {24} {65} {47}

(1) {36 45} {18 53} {30 72} {48 93} {15 24} {47 65}

(2) {18 36 45 53} {30 48 72 93} {15 24 47 65}

(3) {18 30 36 45 48 53 72 93} {45 24 47 65}

(4) {15 18 30 36 45 47 48 53 55 72 93}

图 9-9 归并排序示意图

要得到二路归并的排序算法, 首先要给出一趟归并排序的算法: 设数组  $A(1:n)$  中每个有序表的长度为  $L$  (但最后一个表的长度可能小于  $L$ )。进行两两归并后的结果存于数组  $R(1:n)$  中。进行一趟归并排序时, 对于  $A$  中可能除最后一个 (当  $A$  中有序表个数为奇数时) 或两个 (当  $A$  中有序表个数为偶数, 但最后一个表的长度小于  $L$  时) 有序表, 共剩有偶数个长度为  $L$  的有序表, 由前到后对每两个有序表调用 merge 过程即可完成归并; 对可能剩下的最后两个有序表 (后一个长度小于  $L$ , 否则不会剩下), 继续调用 merge 过程即可完成归并; 对可能剩下的最后一个有序表 (其长度小于等于  $L$ ), 则把它直接复制到  $R$  中对应区间即可。至此, 一趟归并完成。

二路归并排序的过程需要进行  $\log_2 n$  趟。第一趟  $L$  等于 1, 以后每进行一趟将  $L$  加倍。假定待排序的  $n$  个记录保存在数组  $A(1:n)$  中, 归并过程中使用的辅助数组为  $R(1:n)$ , 第一趟由  $A$  归并到  $R$ , 第二趟由  $R$  归并到  $A$ , 如此反复进行, 直到  $n$  个记录成为一个有序表为止。在归并过程中, 为了将最后的排序结果仍置于数组  $A$  中, 需要进行的趟数为偶数, 如果实际只需奇数趟 (即  $\log_2 n$  为奇数) 完成, 那么最后还要进行一趟, 正好此时  $R$  中的  $n$  个有序元素为一个长度不大于  $L$  (此时  $L \geq n$ ) 的表, 将会被直接复制到  $A$  中。二路归并排序的程序较为复杂, 请同学们学习之后自己编写。

二路归并排序的时间复杂度等于归并趟数与每一趟时间复杂度的乘积。归并趟数为  $\log_2 n$ 。因为每一趟归并就是将两两有序表归并, 而每一对有序表归并时, 记录的比较次数和移动次数 (即由一个数组复制到另一个数组中的记录个数) 均等于这一对有序表的长度之和, 所以每一趟归并的比较次数和移动次数均等于数组中记录的个数  $n$  (有一点例外是: 当待归并数组中的有序表个数为奇数时, 最后一个有序表只复制不比较), 亦即每一趟归并的时间复杂度为  $O(n)$ 。因此, 二

路归并排序的时间复杂度为  $O(n \log_2 n)$ 。

二路归并排序时需要利用同待排序数组一样大小的一个辅助数组，所以其空间复杂度为  $O(n)$ 。显然它高于前面所有排序算法的空间复杂度。

二路归并排序是稳定的，因为在每两个有序表归并时，若分别在两个有序表中出现相同排序码的元素，merge 算法能够使前一有序表中同一排序码的元素先被复制。后一有序表中同一排序码的元素后被复制，从而确保它们的相对次序不会改变。

最后还需要指出：“归并”技术不仅适用于内排序，而且更适用于外排序。在外排序中，首先按照内存可使用存储空间的大小，将外存上含有  $n$  个记录的文件分成若干个子文件，使得每个子文件能够一次装入内存中；接着接序处理每一个子文件，处理过程是：读入一个子文件到内存，利用内排序方法进行排序，把排序后的有序子文件（称为初始归并段）写入外存；然后对这些初始有序子文件，利用二路或多路归并技术进行每一趟归并，使有序子文件的个数逐渐减少，长度逐渐增加，直到最后变为一个有序文件为止。当然在每两个或多个有序子文件归并为一个有序子文件的过程中，每次从每个有序子文件中只能读入一批记录而不是全部记录到该文件的内存缓冲区中（一次读入记录的多少，视缓冲区大小而定），当归并完后再接着读入下一批记录；归并结果也是依次放入结果文件的内存缓冲区中，并每当缓冲区满后写入外存，最后在外存得到每次归并后的有序子文件。

## 9.6 各种排序方法比较

各种排序方法之间的比较，主要从以下几个方面综合考虑：①时间复杂度；②空间复杂度；③稳定性；④算法简单性；⑤待排序记录数  $n$  的大小；⑥记录本身信息量的大小。

下面先从每个方面进行比较和分析，然后再给出综合结论。

1. 从时间复杂度看，直接插入排序、直接选择排序和气泡排序这三种简单排序方法属于一类，其时间复杂度为  $O(n^2)$ ；堆排序、快速排序和归并排序这三种排序方法属于第二类，其时间复杂度为  $O(n \log_2 n)$ 。若从最好情况考虑，则直接插入排序和气泡排序的时间复杂度最好，为  $O(n)$ ，其他算法的最好情况同平均情况相同；若从最坏情况考虑，则快速排序的时间复杂度为  $O(n^2)$ ，直接插入排序和气泡排序虽然平均情况下相同，但系数大约增加一倍，所以运行速度将降低一半，最坏情况对直接选择排序、堆排序和归并排序影响不大。若再考虑各种排序算法的时间复杂度的系数，则在第一类算法中，直接插入排序的系数最小，直接选择排序次之（但它的移动次数最小），气泡排序最大，所以直接插入排序和直接选择排序比气泡排序速度快；在第二类算法中，快速排序的系数最小，堆排序和归并排序次之，所以快速排序比堆排序和归并排序速度快。由此可知，在最好情况下，直接插入排序和气泡排序最快；在平均情况下，快速排序最快；在最坏情况下，堆排序和归并排序最快。

2. 从空间复杂度看，所有排序方法可归为三类：归并排序单独属于一类，其空间复杂度为  $O(n)$ ；快速排序也单独属于一类，其空间复杂度为  $O(\log_2 n)$ （但在最坏情况下为  $O(n)$ ）；其他排序方法归为第三类，其空间复杂度为  $O(1)$ 。由此可知，第三类算法的空间复杂度最好，第二类次之，第一类最差。

3. 从稳定性看，所有排序方法可分为两类：一类是稳定的，它包括直接插入排序、气泡排序



和归并排序；另一类是不稳定的，它包括希尔排序（可以参考其他的数据结构书籍）、直接选择排序、快速排序和堆排序。

4. 从算法简单性看，一类是简单算法，它包括直接插入排序、直接选择排序和气泡排序，这些算法都比较简单和直接；另一类是改进后的算法，它包括希尔排序、堆排序、快速排序和归并排序（归并排序可看作为对直接插入排序的另一种改进，它把记录分组排序，但分组方法同希尔排序不同，另外，它把记录的插入和移动改为向另一个数组的复制），这些算法都比较复杂。

5. 从待排序的记录数  $n$  的大小看， $n$  越小，采用简单排序方法越合适， $n$  越大采用改进排序方法越合适。因为  $n$  越小， $O(n^3)$  同  $O(n \log_2 n)$  的差距越小，并且简单算法的时间复杂度的系数均小于 1（除气泡排序中最坏情况外），改进算法的时间复杂度的系数均大于 1，因而也使得它们的差距变小，另外，输入和调试简单算法比输入和调试改进算法要少用许多时间。若把此时间也考虑进去，当  $n$  较小时，选用简单算法比选用改进算法要少花时间。当  $n$  越大时选用改进算法的效果就越显著，因为  $n$  越大， $O(n^2)$  同  $O(n \log_2 n)$  的差距就越大。例如，当  $n=10000$  时， $O(n \log_2 n)$  只是  $O(n^2)$  的约  $1/700$ 。

6. 从记录本身信息量的大小看，记录本身的信息量越大，表明占用的存储字节数就越多，移动记录时所花费的时间就越多，所以对记录的移动次数较多的算法不利。例如，在三种简单排序算法中，直接选择排序移动记录的次数为  $O(n)$  数量级，其他两种为  $O(n^2)$  数量级。所以当记录本身的信息量较大时，对直接选择排序算法有利，而对其他两种算法不利。在四种改进算法中，记录本身信息量的大小，对它们影响区别不大。

以上从六个方面对各种排序方法进行了比较和分析，那么如何在实际的排序问题中分主次地考虑它们呢？首先考虑排序对稳定性的要求，若要求稳定，则只能在稳定方法中选取，否则可以从所有方法中选取；其次要考虑待排序记录数  $n$  的大小，若  $n$  较大，则在改进方法中选取，否则在简单方法中选取；然后再考虑其他因素。

下面给出综合考虑以上六个方面所得出的大致结论，供读者选择内排序方法时参考：

1. 当待排序记录数  $n$  较大，排序码分布较随机，且对稳定性不作要求时，则采用快速排序为宜。
2. 当待排序记录数  $n$  较大，内存空间允许，且要求排序稳定时，则采用归并排序为宜。
3. 当待排序记录数  $n$  较大，排序码分布可能会出现正序或逆序的情况，且对稳定性不作要求时，则采用堆排序（或归并排序）为宜。
4. 当待排序记录数  $n$  较小（如小于 100），记录或基本有序（即正序）或分布较随机，且要求稳定时，则采用直接插入排序为宜。
5. 当待排序记录数  $n$  较小，对稳定不作要求时，则采用直接选择排序（若排序码不接近逆序，亦可选用直接插入排序）为宜。

在信息学竞赛中，我们最常使用的排序方法是快速排序。但如果数据范围不大，我们也可以选择较易编写的选择排序。堆往往是作为一种数据结构使用。

## 9.7 线性时间排序

到目前为止，所讨论的排序算法有一个共同的特点，即用于得到确定排序结果的主要运算是输入元素间的比较运算。这类排序算法称为基于比较的排序算法。本节讨论三个以数字和地址计

算为主要运算的排序算法：计数排序、桶排序和基数排序。由于这些算法已不再是基于比较的排序算法，所以  $O(n \log_2 n)$  计算时间下界的复杂度对它们已不适用；事实上，它们都可以在线性时间内完成排序任务，但是它们的应用都有一定的局限性。

## 一、计数排序

计数排序算法的基本思想是：对于给定的输入序列中的每一个元素  $x$ ，确定该序列中关键字小于  $x$  的元素个数。一旦有了这个信息，就可以将  $x$  直接存放到最终的输出序列的正确位置上。例如，如果输入序列中只有 17 个元素的关键字小于  $x$  的关键字，则  $x$  可以直接存放在输出序列的第 18 个位置上。当然，如果有多个元素具有相同的关键字时，我们不能将这些元素放在输出序列的同一位置上，因此上述方案还要作适当的修改。

在下面的计数排序算法中，假设输入的  $n$  个元素存放在数组  $A[1..n]$  中，输出的排序结果存放在数组  $B[1..n]$  中。数组  $A$  和  $B$  中的每个元素是一个记录，其类型为 `recordtype`，其中关键字的类型为 `keytype`。在本节中，设为有限类型，如 `1..m` 或 `char` 等。为明确起见，取 `keytype` 的类型为 `1..m`，即所有输入元素的关键字的值是 1 到  $m$  之间的一个整数。算法中还要用到一个辅助数组  $C[1..m]$  用于对输入元素进行计数。则计数排序的过程为：

```

Procedure Countsort (Var a, b: array [1..m] of recordtype);
Var
    I, j: integer;
    C: array [1..m] of integer;
Begin
    For I := 1 to m do c[I] := 0;
    For j := 1 to n do c[a[j].key] := c[a[j].key] + 1;
    {将输入序列中关键字等于 I 的元素个数存放在 C[I] 中}
    For I := 2 to m do c[I] := c[I] + c[I-1];
    {C[I] 中存放了输入序列中关键字值小于或等于 I 的元素的个数}
    For j := n downto 1 do
        Begin
            B[c[a[j].key]] := a[j];
            C[a[j].key] := c[a[j].key] - 1;
        End
    End;
End;
```

计数排序的计算时间的复杂度很容易分析，整个算法的时间复杂度为  $O(m+n)$ ，当  $m = O(n)$  时，算法的时间复杂度为  $O(n)$ 。计数排序算法没有用到元素间的关键字的比较，它利用实际的关键字的值来确定它们在数组中的位置，因此计数排序不是一种基于比较的排序算法。由于对它的关键字的取值范围进行了限定，所以在时间复杂度上取得了线性的优势。通过计数排序算法可以看出，它是一种稳定的排序算法。

## 二、桶排序

与计数排序类似的一个线性时间排序算法是桶排序算法。其基本思想是：设置若干个桶，将

关键字等于  $i$  的元素全部装入第  $i$  桶中, 然后按桶的顺序将桶中的元素顺序连接起来。由于每个桶中都有相同的关键字, 可以将第  $i$  桶看作是关键字为  $i$  的元素组成的一个表。若设表的类型为 `Listtype`, 用数组 `B` 表示桶序列, 则 `B` 的类型为 `array[keytype] of Listtype`。如果用链表来实现表, 则每个桶 `B[i]` 就是表头。要将桶 `B[i]` 中的表  $L_i$  和桶 `B[j]` 中的表  $L_j$  连接起来, 可用表的连接运算, 它将表  $L_i$  的内容改变为  $L_i L_j$ 。

为节省时间, 可为每个表设置一个指向表尾的指针, 这样, 当要找表中最后一个元素时, 就不必将表整个地扫描一遍。图 9-10 说明如何将表  $L_i$  与  $L_j$  连接起来得到一个新表, 且新表的名字为  $L_i$ 。其中虚线表示指针变化。在连接运算结束后, 表  $L_j$  就变成空表了。

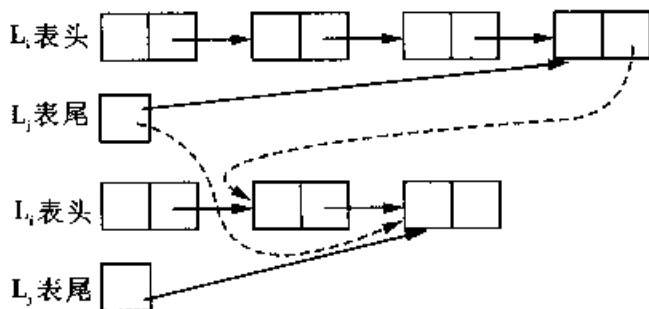


图 9-10 链表的链接

下面给出桶排序的算法。算法是用表的基本运算写出的, 其中, 假设关键字类型 `keytype` 为  $1..m$ , 即关键字是在 1 到  $m$  范围内的一个整数。输入数组为 `A: array[1..n] of recordtype`, 桶数组为 `B: array[keytype] of listtype`。

Procedure BUCKETSORT;

Var

I: integer;

J: keytype;

Begin

桶数组初始化;

For i: =1 to n do {将元素装入桶中}

INSERT (A[i], FIRST (B[A[i].key]), B[A[i].key]);

For j: =2 to m do {将所有桶连接到 B[1] 后面}

CONCATENATE (B[1], B[j])

End;

桶排序算法所需的计算时间与计数排序算法大致相同, 也是  $O(m+n)$ 。与计数排序类似, 当  $m=O(n)$  时, 则桶排序的时间复杂度也是  $O(n)$ 。

### 三、基数排序

在计数排序和桶排序算法中, 如果  $m=n^2$ , 则算法需要的计算时间不再是  $O(n)$ , 而是  $O(m+n)=O(n^2)$ 。这是由于关键字的取值范围扩大所造成的。然而, 如果关键字的取值范围是有限的。比如  $m \leq n^k$ ,  $k$  是预先确定的正整数, 那么是否有保持  $O(n)$  时间复杂度的排序算法? 事实上, 利用  $m$  的有限性, 并以桶排序为工具, 设计出一个新的排序算法。

先来考虑一个特殊情形, 即对  $0 \dots n^2 - 1$  范围内的  $n$  个整数进行排序的问题, 分两步来完成排序工作。第一步用  $n$  个标号分别为  $0, 1, \dots, n-1$  的桶, 将整数  $i$  装入标号为  $i \bmod n$  的桶中。注意, 这里所说的将一个元素装入一个桶中是指将这个元素接到桶中元素表的末尾。为了有效地进行这样的操作, 要求桶中的链接表具有指向表尾的指针。

例如, 设  $n=10$ , 待排序的整数序列是有机排列的  $0 \dots 99$  范围内的完全平方数  $36, 9, 0, 25, 1, 49, 64, 16, 81, 4$ 。此时, 桶的标号  $i \bmod n$  正好是十进制整数  $i$  的个位数。图 9-11 (a) 说明了第一次将这些整数装入桶中的情况。

注意在第一次将整数装入桶中时, 每个桶中各整数的顺序保持了它们在输入时的相对次序。例如, 第 6 号桶中装的是  $36, 16$ , 而不是  $16, 36$ , 因为在输入时  $36$  在  $16$  的前面。

第一次将所有整数装入桶中后, 将各桶中整数顺序连接起来。例如, 从图 9-10 (a) 得到连接后的整数序列为:  $0, 1, 81, 64, 4, 25, 36, 16, 9, 49$ 。

显然, 此时各整数的个位数排成了递增顺序。由于在各桶中设置了指向表尾指针, 上述过程共用  $O(n)$  时间。

第二步仍使用这  $n$  个桶, 标号还是  $0, 1, \dots, n-1$ 。按照第一步得到的各整数的排列次序, 将这些整数重新装入桶中。这一次是将整数  $i$  装入标号为  $i \div n$  的桶中, 新装入的数仍然接在表尾。最后, 再将各桶中的整数顺序连接在一起, 所得到的结果就是排好序的整数序列。

对于上述  $n=10$  的例子,  $i \div n$  恰好是十进制数的十位数。因此, 第二次装入桶中的整数如图 9-11 (b) 所示。

桶	内容	桶	内容
0	0	0	0, 1, 4, 9
1	1, 81	1	16
2		2	25
3		3	36
4	64, 4	4	49
5	25	5	
6	36, 16	6	64
7		7	
8		8	81
9	9, 49	9	
(a)		(b)	

图 9-11 元素两次装入桶中的情况

下面讨论算法的正确性。

对于上述  $n=10$  的例子, 在第一次装桶后, 各整数的个位数字已排成了递增顺序, 从而第二次装桶后, 每个桶中的各整数十位数相同, 且个位数递增, 因此同一桶中的整数已排成递增顺序。又由于装入桶号小的桶中的十位数小, 桶号大的桶中的十位数大, 所以顺序连接各桶后, 各整数就排成了递增顺序。

对于一般的  $n$ , 可将  $0 \dots n^2 - 1$  范围内的每一个整数看成两位的  $n$  进制整数。算法还是上述两

步, 正确性证明类同。事实上, 设  $i = an + b$ ,  $j = cn + d$ , 其中  $a, b, c, d$  都是 0 到  $n-1$  范围内的整数, 它们可看成是  $n$  进制整数的一位数字。设  $i < j$ , 那么必有  $a \leq c$ 。如果  $a < c$ , 那么在第二次装桶时,  $i$  被装入桶号小的桶中,  $j$  被装入桶号大的桶中。这样在将各桶中整数连接起来时,  $i$  排在  $j$  前面。如果  $a = c$ , 必有  $b < d$ 。那么, 在第一步完成之后,  $i$  排在  $j$  前面。在第二次装桶时,  $i$  和  $j$  都被装入第  $a$  个桶中, 并且  $i$  仍然排在  $j$  前面。这样, 在第二步完成之后,  $i$  照样排在  $j$  前面。由此可知, 在算法的两步都完成以后, 所有整数已排好序。

上述算法的思想可推广到更一般的情形。例如 `keytype` 是由若干个域组成的记录的情形:

```
type
    keytype = record
        day: 1..31;
        month: (jan..dec);
        year: 1900..1999;
    end;
```

和 `keytype` 是由某个类型的元素组成的数组的情形:

```
type
    keytype = array [1..10] of char;
```

更一般地, `keytype` 可以由  $k$  个分量  $f_1, f_2, \dots, f_k$  所组成。其中  $f_i$  的类型是  $t_i$  ( $1 \leq i \leq k$ )。

在最后一种情形下, 要将输入元素按其关键字的字典序进行排序。按照关键字的字典序, 关键字  $(a_1, a_2, \dots, a_k)$  小于关键字  $(b_1, b_2, \dots, b_k)$  的充要条件是下述  $k$  个条件之一成立:

- (1)  $a_1 < b_1$ ;
- (2)  $a_1 = b_1, a_2 < b_2$ ;
- .....
- (k)  $a_1 = b_1, a_2 = b_2, \dots, a_{k-1} = b_{k-1}, a_k < b_k$ 。

即, 存在 0 到  $k-1$  范围内的一个整数  $j$ , 使得  $a_1 = b_1, \dots, a_j = b_j$  而  $a_{j+1} < b_{j+1}$ 。

可以将上述类型的关键字看作是以特殊的基数来表示的整数。在一般情况下, `keytype` 中的  $k$  个分量  $f_1, f_2, \dots, f_k$  组成数的  $k$  个位,  $f_i$  的基数为它的取值范围  $l_i$ ,  $i = 1, 2, \dots, k$ 。例如, 当 `keytype = array [1..10] of char` 时,  $k = 10$ ,  $f_i$  是计算机定义的字符集中的任意一个字符,  $l_i$  是该字符集中字符的个数  $l$ , 比如 128。Keytype 变量可看成是一个 10 位的  $l$  进制的整数。在这种观点下推广的桶排序算法就称为基数排序算法。

基数排序算法的基本思想是: 首先对  $n$  个输入元素按  $f_k$  进行桶排序 (在决定元素关键字的大小时,  $f_k$  的作用最小), 然后按  $f_{k-1}$  进行桶排序……与上面的例子一样, 每次将元素装入桶中时, 总是将所装入的元素接在表尾。一般的基数排序算法可描述如下:

Procedure READXSORT; {对  $n$  个输入元素组成的表  $A$  进行排序, 每个元素的关键字由域  $f_1, f_2, \dots, f_k$  组成,  $f_i$  的类型为  $t_i$ 。算法中用到桶数组  $B_i$ : `array [ti] of listtype`,  $1 \leq i \leq k$ , 其中 `listtype` 是由元素组成的链接表}

```
Begin
    For i := k downto 1 do
        Begin
```

```

For 类型  $t_i$  的每个值  $v_i$  do {清桶}
    置  $B_i[v_i]$  为空表
For 表 A 中的每个元素  $r$  do
Begin
     $v_i := r.f_i$ 
    将  $r$  装入桶  $b_i[v_i]$  中
End;
置 A 为空;
For 类型  $t_i$  的每个值  $v_i$ , 从小到大的 do
    将  $b_i[v_i]$  连接到 A 后面
End
End {READXSORT}

```

与上面所讨论  $k=2$  时的基数排序算法类似, 可以证明一般的基数排序算法的正确性, 即分别依  $f_k, f_{k-1}, \dots, f_1$  对 A 进行桶排序后, 所有输入元素按  $f_1, f_2, \dots, f_k$  的字典序排列。

选用合适的数据结构, 可使基数排序算法节省时间。为此用链表形式表示输入序列, 而不用数组。这样很容易将一个表移到另一个表中。如果输入元素是用数组 A 表示的, 只要在记录中增加一个指针域, 使  $A[i]$  的指针指向  $A[i+1]$ , 就可将数组 A 改成链接表, 这只需要  $O(n)$  的时间。

## 9.8 排序的应用举例

在竞赛中很少单独一道题指明要求排序, 一般是在解题过程中需要用到排序过程或通过对问题的分析后确定用排序算法解决。下面通过几个具体实例来讨论排序算法的应用。

**例题 9-2** 由文件给出  $N$  个 1 到 30000 间无序正整数, 其中  $1 \leq N \leq 10000$ , 同一个正整数可能会出现多次, 出现次数最多的整数称为众数。求出它的众数及它出现的次数。

分析:

该题实际上是一道统计正整数出现次数的题目, 统计出次数后再求出它们次数的最大值, 联系到学过的排序法, 自然会想到利用桶排序的思想, 设计出如下算法:

- S1: 定义一个桶数组  $A[1..30000]$ , 类型为整型, 初始化为 0;
- S2: 将提供文件中的  $N$  个整数分别装入以下标为编号的对应桶中;
- S3: 求桶数组 A 的所有最大值 (可能有多个) 并输出;
- S4: 结束。

类似的题如: 有杂乱排列的  $N$  个正整数, 求出所有的最大平台 (即出现次数最多的数) 的平台宽度 (最大平台中, 数的个数) 和平台高度 (最大平台中, 数的值)。

**例题 9-3** 修建输油管道。

问题描述:

某石油公司计划建造一条由东向西的主输油管道。

该管道要穿过一个  $n$  口油井的油田。从每口油井都要有一条输油管道沿最短路径 (或南或北) 与主管道相连, 如图 9-12 所示。如果给定  $n$  口油井的位置, 即它们的 X 坐标和 Y 坐标, 应如何

确定主管道的最优位置，即使各油井到主管道之间的输油管道长度总和最小的位置？

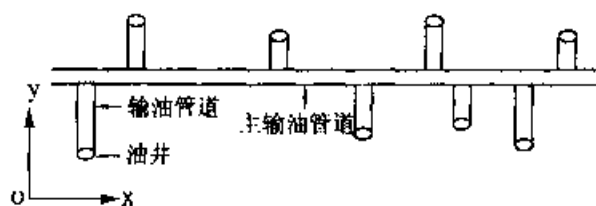


图 9-12 输油管道布局

分析：

初看此道题似乎与最短距离问题有关，但好像又无从下手，实际上仔细分析后，问题就会变得相当简单。设  $n$  口油井的位置分别为  $P_i = (X_i, Y_i)$ ,  $i = 1, 2, \dots, n$ ；由于主管输油管道是东西方向的，因此可用其主轴线的  $Y$  坐标唯一确定其位置，主管道的最优位置  $Y$  应使  $\sum_{i=1}^n D(Y, Y_i)$  达到最小，其中  $D(Y, Y_i) = |Y - Y_i|$ 。这实际上就是  $n$  口油井的  $Y$  坐标  $Y_1, Y_2, \dots, Y_n$  中找出它们的中位数  $Y_k$ ，就是主管输油管道的最优位置，用任何一个线性时间  $O(n)$  的时间复杂度就能够解决的问题。这样的题容易，但算法具有一定的隐秘性。

例题 9-4 打水问题。

问题描述：

有  $n$  个人在一个水龙头前排队接水，每个人接水的时间  $T_i$  是互不相等的。找到一种这  $n$  个人排队接水的顺序，使他们平均等待的时间达到最小。

分析：

这是一道数学国际奥赛试题，一看就是一个组合最优化问题，但千万不要用动态规划去做，实际上是贪心算法的典型应用，最后归结为一个排序。平均等待时间是每个人的等待时间之和，再除以  $n$ 。因为  $n$  是一个常数，所以等待时间最小，也就是平均等待时间最小。因此，这个问题就是求函数：

$S = T_1 + (T_1 + T_2) + (T_1 + T_2 + T_3) + \dots + (T_1 + T_2 + T_3 + \dots + T_n) = \sum_{i=1}^n (\sum_{j=1}^i T_j)$  的最小值，这个函数称它为目标函数。

分析目标函数，它可改写成如下形式：

$$S = nT_1 + (n-1)T_2 + \dots + 2T_{n-1} + T_n = \sum_{j=1}^n (n-1+j)T_j$$

仔细研究该式的特点，可以得到如下定理：

对于  $n$  个人的任一种排列  $k_1, k_2, \dots, k_n$ ，若有  $T_{k_1} \leq T_{k_2} \leq T_{k_3} \leq \dots \leq T_{k_n}$ ，则目标函数  $S$  取得最小值。该定理的证明很简单，采用反证法很容易得到该结论。

根据这一定理，就可以得到一种贪心的最优解法：只要把  $n$  个人等待时间按非递减的顺序排列，即把  $n$  个人接水的时间按递减的顺序排列，这时平均等待时间可以达到最小。这样，求最小值的问题就转化为一个简单的排序问题。算法如下：

Procedure watersort ( $n, t, s$ )

Begin

输入  $n$  和  $T[1], T[2], \dots, T[n]$ ；并保存  $n$  个人的编号数组  $B$ ；

把数组  $T$  从小到大排序；同时  $B$  数组随  $T$  数组变化；

输出 B 数组中的编号;

计算 S 值和输出 S 值;

End;

类似的题如:有  $N$  个人排队到  $R$  个相同的水龙头上接水,他们接满水桶的时间  $T_1, T_2, \dots, T_n$  为整数且各不相同,应如何安排他们接水的顺序才能使他们接水花费的总时间最小?请给出一种具体安排,并求出花费的最小时间。

**例题 9-5** 求第  $k$  小整数。

问题描述:

现有  $n$  个正整数,  $n \leq 10000$ , 要求出这  $n$  个正整数中的第  $k$  个最小整数(相同大小的整数只计算一次),  $k \leq 1000$ , 正整数均小于 65535。

输入:第一行为  $n$  和  $k$ , 第二行开始为  $n$  个正整数的值, 整数间用空格隔开。

输出:第  $k$  个最小元素的值;若无解,则输出“NO RESULT”。

输入输出示例: INPUT1.TXT                  OUTPUT1.TXT

10 3                                  3

1 3 3 7 2 5 1 2 4 6

分析:

该题要求  $n$  个数中的第  $k$  个最小数,相同的数只计算一次,而并没有要得到所有数的在  $n$  个数中顺序,所以不一定用排序方法,但排序肯定能得到所要求的信息。显然可以用快速排序思想中的分法策略可求得问题的解,这里从排序方面来考虑。

从排序方面考虑,由于数据量大,采用时间复杂度为  $O(n^2)$  的排序算法要在规定时间内出解有困难,因此可采用时间复杂度为  $O(N \log_2 n)$  的快速排序和堆排序算法。这里还有一种利用桶排序的较好算法。

设一个从 1 到 65535 的布尔数组,用来记录数据当中所出现的不同的数值。由于问题当中讲到相同大小的整数只计算一次,所以可以从该数组当中顺序查找出现的第  $k$  个数值,即为所求。该算法的时间复杂度为  $O(N)$ 。程序流程为:在读入数据同时,记录数据当中出现的不同数值。再从 1 开始顺序查找出现的第  $k$  个数值,并将其输出,若不存在第  $k$  个数值,则输出“NO RESULT”。源程序如下:

Program ex9-5;

Const

inp = 'input.txt';

out = 'output.txt';

Type

list = array [1..32767] of boolean;

Var i, j, k, n, p: word;

a: array [1..2] of ^list; {标志数组}

df: text;

Procedure Print; {输出过程}

Begin

Assign (df, out);



```

    Rewrite (df);
    if p=1 then Writeln (df, i) |当求出的解小于 32767 时|
        else Writeln (df, i+32767); |当求出的解大于 32767 时|
    Close (df);
    halt
End;
Begin
    Assign (df, inp);
    Reset (df);
    Readln (df, n, k);
    New (a [1]);
    New (a [2]);
    for i: = 1 to 32767 do |初始化|
        begin
            a [1] ^ [i]: = false;
            a [2] ^ [i]: = false
        end;
    for i: = 1 to n do |标志数组赋值|
        begin
            Read (df, j);
            if (j>32767) and (j mod 32767 <> 0) then a [2] ^ [j mod 32767]: = true
                else if j<32767 then a [1] ^ [j]: = true
                    else a [2] ^ [32767]: = true
        end;
    Close (df);
    j: = 0;
    for i: = 1 to 32767 do |求解过程|
        begin
            if a [1] ^ [i] then inc (j);
            if j=k then begin p: = 1; print end
        end;
    for i: = 1 to 32767 do
        begin
            if a [2] ^ [i] then inc (j);
            if j=k then begin p: = 2; print end;
        end;
    if j < k then begin |无解|
        Assign (df, out);
        Rewrite (df);

```

```

        Writeln (df, 'NO RESULT');
        Close (df)

    end

End.
    
```

## 9.9 小结

在这一章比较全面地学习了排序的各种方法,分析了他们的复杂度及适用的范围。全面掌握排序的方法在编程中是相当重要的。我们在解决问题时大多要用到排序的算法,在实现某些算法时必须要先进行排序,同时排序为了更快地实现查找与处理,因此可以将排序归结为一种基础的算法。基础的重要性是不言而喻的。

## 习题九

### 一、单选题

1. 若对  $n$  个元素进行直接插入排序,在进行第  $i$  趟 ( $1 \leq i \leq n-1$ ) 排序时,为寻找插入位置最多需要进行( )次元素的比较。  
A.  $i+1$                       B.  $i-1$                       C.  $i$                           D. 1
2. 若对  $n$  个元素进行直接插入排序,在进行任一趟排序的过程中,寻找插入位置的时间复杂度为( )。  
A.  $O(1)$                       B.  $O(n)$                       C.  $O(n^2)$                       D.  $O(\log_2 n)$
3. 在对  $n$  个元素进行快速排序的过程中,第一次划分最多需要交换( )对元素。  
A.  $n/2$                           B.  $n-1$                           C.  $n$                               D.  $n+1$
4. 在对  $n$  个元素进行快速排序的过程中,最好情况下需要进行( )层划分。  
A.  $n$                               B.  $n/2$                           C.  $\log_2 n$                           D.  $2n$
5. 在对  $n$  个元素进行直接选择排序的过程中,需要进行( )趟选择和交换。  
A.  $n$                               B.  $n+1$                           C.  $n-1$                           D.  $n/2$
6. 若对  $n$  个元素进行直接选择排序,则进行任一趟排序的过程中,寻找最小值元素的时间复杂度为( )。  
A.  $O(1)$                           B.  $O(\log_2 n)$                       C.  $O(n^2)$                           D.  $O(n)$
7. 若对  $n$  个元素进行堆排序,则在构成初始堆的过程中需要进行( )次筛选运算。  
A. 1                                  B.  $n/2$                           C.  $n$                               D.  $n-1$
8. 若对  $n$  个元素进行堆排序,则在由初始堆进行每趟排序的过程中,共需要进行( )次筛选运算。  
A.  $n+1$                           B.  $n/2$                           C.  $n$                               D.  $n-1$
9. 若对  $n$  个元素进行归并排序,则进行归并的趟数为( )。  
A.  $n$                                   B.  $n-1$                           C.  $n/2$                           D.  $\log_2 n$

10. 下列排序算法中, 每一趟都能选出一个元素放在其最终位置上, 并且是不稳定的排序算法是( )。

- A. 冒泡排序  
B. 希尔排序  
C. 直接选择排序  
D. 直接插入排序

### 二、填空题

1. 每次从无序表中取出一个元素, 把它插入到有序表中的适当位置, 此种排序方法叫做\_\_\_\_\_排序; 每次从无序表中挑选出一个最小或最大元素, 把它交换到有序表的一端, 此种排序方法叫做\_\_\_\_\_排序。

2. 每次直接或通过基准元素间接比较两个元素, 若出现降序排列时就交换它们的位置, 此种排序方法叫做\_\_\_\_\_排序; 每次使两个相邻的有序表合并成一个有序表的排序方法叫做\_\_\_\_\_排序。

3. 在堆排序的过程中, 对  $n$  个记录建立初始堆需要进行\_\_\_\_\_次筛运算; 由初始堆到堆排序结束, 需要对树根结点进行\_\_\_\_\_次筛运算。

4. 在堆排序的过程中, 对任一分支结点进行筛运算的时间复杂度为\_\_\_\_\_, 整个堆排序过程的时间复杂度为\_\_\_\_\_。

5. 假定一组记录的排序码为 (46, 79, 56, 38, 40, 84), 则利用堆排序方法建立的初始堆为\_\_\_\_\_。

6. 快速排序的平均情况下的时间复杂度为\_\_\_\_\_, 在最坏情况下的时间复杂度为\_\_\_\_\_。

7. 快速排序的平均情况下的空间复杂度为\_\_\_\_\_, 在最坏情况下的空间复杂度为\_\_\_\_\_。

8. 假定一组记录的排序码为 (46, 79, 56, 38, 40, 80), 对其进行快速排序的一次划分的结果为\_\_\_\_\_。

9. 假定一组记录的排序码为 (46, 79, 56, 38, 40, 80), 对其进行快速排序的过程中, 对应二叉搜索树的深度为\_\_\_\_\_, 分支结点数为\_\_\_\_\_。

10. 在归并排序中, 进行每趟归并的时间复杂度为\_\_\_\_\_, 整个排序过程的时间复杂度为\_\_\_\_\_, 空间复杂度为\_\_\_\_\_。

11. 对 20 个记录进行归并排序时, 共需要进行\_\_\_\_\_趟归并, 在第三趟归并时是把长度为\_\_\_\_\_的有序表两两归并为长度为\_\_\_\_\_的有序表。

### 三、上机编程题

#### 1. 竞赛排名。

某市组织一次中学生科技全能竞赛, 每个选手要参加数学、物理、天文、地理、生物、计算机和英语共八项竞赛, 最后综合八项竞赛的成绩排出总名次。选手编号依次为 1, 2, …,  $n$  ( $n$  为参加竞赛的总人数)。

设  $x_{ij}$  表示编号为  $i$  的选手第  $j$  项竞赛成绩 ( $1 \leq i \leq N, 1 \leq j \leq 8$ )。其他指标如下:

(1) 第  $j$  项竞赛的平均分:  $avg_j = \frac{1}{N} \sum_{i=1}^N x_{ij} \quad (1 \leq j \leq 8)$

(2) 选手  $i$  的总分:  $sum_i = \frac{1}{N} \sum_{j=1}^8 x_{ij}, \quad (1 \leq i \leq N)$

$$(3) \text{ 选手 } i \text{ 第 } j \text{ 项竞赛的位置分: } y_{ij} = \begin{cases} 0 & (\sum_{i=1}^N |x_{ij} - \text{avg}_j| = 0) \\ \frac{x_{ij} - \text{avg}_j}{\frac{1}{N} \sum_{i=1}^N |x_{ij} - \text{avg}_j|} & (\sum_{i=1}^N |x_{ij} - \text{avg}_j| \neq 0) \end{cases}$$

$$(4) \text{ 选手 } i \text{ 的总位置分: } \text{sum}y_i = \sum_{k=1}^3 y_{ik} + 0.8 \sum_{k=4}^8 y_{ik} \quad (1 \leq i \leq N)$$

排名规则如下:

- (1) 总位置分高的选手名次在前;
- (2) 若两个或两个以上的选手总位置分相同, 则总分高的选手名次在前;
- (3) 若两个或两个以上的选手总位置分和总分均相同, 则编号在前的选手名次在前。

请为竞赛委员会编一个程序, 计算本次全能竞赛的总排名情况。

输入数据: 输入文件为 INPUT.TXT。文件的第一行为参赛总人数  $N$  ( $1 \leq N \leq 1000$ ), 从第二行到第  $N+1$  行依次为编号为 1 到编号为  $N$  的选手的成绩, 每行有 8 个 0 到 100 之间的整数, 代表该选手的 8 项竞赛成绩  $x_{i1}, x_{i2}, \dots, x_{i8}$ 。同一行相邻两数间用空格隔开。

输出数据: 输出文件为 OUTPUT.TXT。文件有  $N$  行, 每行依次为排名第 1 的选手的编号, 排名第 2 的选手编号,  $\dots$ , 排名第  $N$  的选手的编号。

输入输出示例: INPUT.TXT                  OUTPUT.TXT

3	1
82 73 68 95 86 82 90	3
72 90 50 60 80 70 65 80	2
72 82 73 68 95 86 82 90	

2. 求最大平台宽度及高度。

有杂乱排列的  $N$  个正整数, 求出所有的最大平台 (即出现次数最多的数) 的平台宽度 (最大平台中, 数的个数) 和平台高度 (最大平台中, 数的值)。

例如: 输入: 1, 2, 3, 2, 3, 4, 1, 4, 5, 5, 2, 4

则输出:

宽度: 3 高度: 2

宽度: 3 高度: 4

## 10 模拟试题

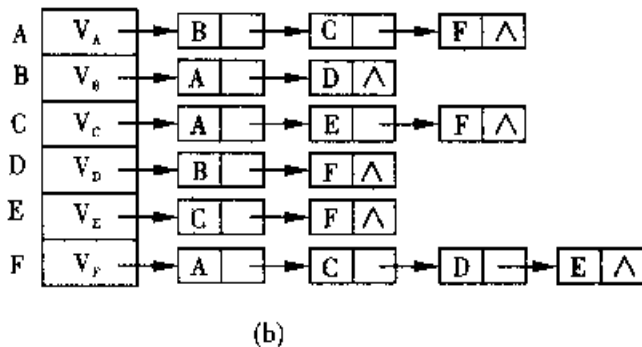
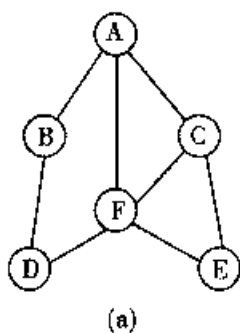
为了巩固知识,加强练习,自己检测知识掌握程度,这里提供给读者四份试题自测,其中两份笔试题和两份上机考试题,并附有参考答案。

### 10.1 数据结构综合测试一

(时量:3小时 总分:100分)

#### 一、填空题(32分)

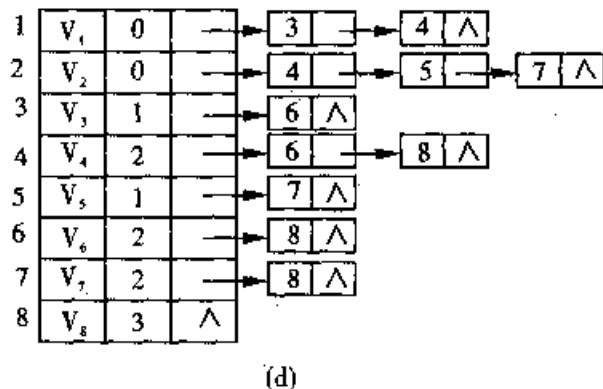
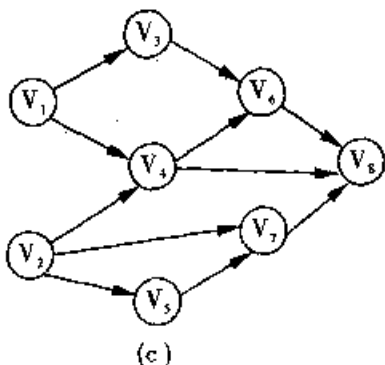
1. 对于线性表的查找,有三种最基本的查找方法,即:( )、( )、( )。
2. 在计算机科学中,算法是描述计算机解决给定问题的操作过程。通常,一个算法必须具备五个重要特性,即:( )、( )、( )、( )、( )。
3. 在线性结构、树型结构和图型结构中,元素之间的联系分别对应为( )、( )和( )。
4. 假定有三个元素 A, B, C 进栈,进栈次序为 ABC,试写出所有可能的出栈序列:( )。
5. 队列是限定所有的插入操作在表的一端进行,而删除操作在表另一端进行的线性表。它操作的是按( )原则进行的。用数组  $q[1..m]$  来存储队列,为了指示队首和队尾,需引进两个指针:F——指向实际队首元素的前一个位置,R——指向实际队尾元素所在位置;循环队列,当  $R = ( )$  后,一旦  $R = ( )$ ,则为队满;出队时,当  $R = ( )$ ,则为空排。
6. 中缀算术表达式  $3 * (5 + x) / y - z$  所对应的前缀算术表达式和后缀表达式分别为:( )和( )。
7. 已知一个图如下图(a)所示,它的邻接表如下图(b)所示,试写出从 VA 出发分别按深度优先遍历和广度优先遍历得到的顶点序列。



深度优先遍历:

广度优先遍历:

8. 已知一个图如下图 (c) 所示, 它的邻接表如图 (d) 所示, 试写出按照拓扑排序算法得到的拓扑序列。



## 二、证明题 (8 分)

如果一棵树有  $n_1$  个度数为 1 的结点,  $n_2$  个度数为 2 的结点, …… $n_m$  个度数为  $m$  的结点, 则该树终端结点的个数  $n_0 = 1 + \sum_{i=1}^m (i-1) * n_i$ 。

## 三、程序填空题 (60 分)

1. [问题描述] 对一个稀疏矩阵  $M[x, y]$ , 求它的转置矩阵  $N[y, x]$ 。

[算法分析] 对稀疏矩阵采用三元组  $(i, j, val)$  的形式贮存, 矩阵  $M$  按行优先的顺序存储于数组  $A$  中, 转置后的矩阵  $N$  也按行优先的顺序存储在数组  $B$  中, 显然数组  $B$  是数组  $A$  经过变换后得到的, 题目要求就是根据  $A$  求  $B$ 。以下算法就是按照矩阵  $M$  的行序进行转置, 具体方法如下:

① 求出矩阵  $M$  中每一列非零元素的个数, 贮存在数组  $num[1..x]$  中, 而矩阵  $M$  中第一列第一个非零元素转置后必放在  $B[1]$  中, 这样就可以推算出矩阵  $M$  中每一列第一个非零元素在数组  $B$  中的位置, 用数组  $put[1..x]$  来贮存。显然有:

$$\begin{cases} put[1] = 1 \\ put[col] = put[col-1] + num[col-2] \quad (2 \leq col \leq n) \end{cases}$$

② 当某一列中转置了一个非零元素后, 就修改  $put[col]$ , 这样使用  $put$  数组就可以确定矩阵  $M$  每一个非零元素转置后在数组  $B$  中的位置。

[程序说明]

Const tal = 20; nmax = 10;

Type node = record

i, j, val: integer;

end;

listar = array [0..tal] of node;

Procedure transl (var a, b: listar);

Var num, put: array [1..nmax] of integer;

col, n, k: integer; t: 0..tal;

Begin

```

n := a [0] . j; t := a [0] . val;
with b [0] do
begin ①; ②; val := a [0] . val; end;
if ③ then
begin
  for col := 1 to n do num [col] := 0;
  for k := 1 to t do num [a [k] . j] := num [a [k] . j] + 1;
  put [1] := 1;
  for col := 2 to n do put [col] := put [col - 1] + num [col - 1];
  for k := 1 to t do
  begin
    ④;
    b [put [col]] . i := a [k] . j; b [put [col]] . j := a [k] . i;
    b [put [col]] . val := a [k] . val;
    ⑤;
  end;
end;
End;

```

2. [问题描述] 给出一组权值, 构造一棵哈夫曼树。

[算法分析] ①设给定的一组权值为  $\{W_1, W_2, \dots, W_n\}$ , 据此生成森林  $F = \{T_1, T_2, \dots, T_n\}$ ,  $F$  中的每一棵树  $T_i$  只有一个带权为  $W_i$  的根结点;

②在  $F$  中选取两棵根结点权值最小的树作为左右子树构造一棵新的二叉树, 新二叉树的根结点的权值为其左右子树的根结点的权值之和;

③在  $F$  中删除这两棵树, 同时将新生成的二叉树并入森林  $F$ ;

④重复②和③, 直到  $F$  中只有一棵树为止。

二叉树的每个结点设四个域:

tag	lc	data	rc
-----	----	------	----

其中, tag 域是标志域;

tag = 0 表示该结点尚未链入二叉树;

tag = 1 表示该结点已经链入二叉树;

lc, rc 域存放左右孩子的地址, data 域存放权值。

[程序说明]

Const nodemax = 50; max = 99;

Type

node = record

tag: 0..1;

```

        data: integer;
        lc, rc: 0..max;
    end;
    rtype = array [1..max] of node;
Procedure huffman (var t: integer; var r: rtype);
Var
    y, n, m1, m2, x1, x2, i, j: integer;
Begin
    writeln ( 'Please input n: ' );
    readln ( n );
    for i: = 1 to n do
        for i: = 1 to nv do
            readln ( y ); r [ i ] . data: = y;
            r [ i ] . tag: = 0; r [ i ] . lc: = 0; r [ i ] . rc: = 0;
        end;
        i: = 0;
        while ① do
            begin
                m1: = 32767; m2: = 32767;
                x1: = 0; x2: = 0;
                for j: = 1 to n + i do
                    begin
                        if ②
                            then begin
                                m2: = m1; x2: = x1; m1: = r [ j ] . data; x1: = j;
                            end
                        else
                            if ③
                                then begin
                                    m2: = r [ j ] . data; x2: = j;
                                end;
                    end;
                end;
                ④ ; ⑤ ;
                i: = i + 1;
                r [ n + i ] . data: = r [ x1 ] . data + r [ x2 ] . data;
                r [ n + i ] . lc: = x1; r [ n + i ] . rc: = x2; r [ n + i ] . tag: = 0;
            end;
            t: = 2 * n - 1;
        End;

```



3. [问题描述] 求网  $G(V, E)$  的最小生成树。

[算法分析] (普里姆算法) 用邻接矩阵来存储网, 网的邻接矩阵通常称为代价矩阵, 一个有  $n$  个顶点的网用一个  $n$  阶方阵  $cost$  表示正权图 (即没有权为 0 和负数的边):

$$cost[i, j] = \begin{cases} w_{ij} & \text{if } (i, j) \in E(G) \vee \langle i, j \rangle \in E(G), i \neq j \\ 0 & \text{if } i = j \\ \infty & \text{else} \end{cases}$$

引入一个概念, 把顶点  $V$  和一集合  $U$  中各顶点所组成的边的权值最小者称为顶点  $V$  到集合  $U$  的距离。

设立两个辅助数组:  $closest[1..n]$  和  $lowcost[1..n]$ 。  $lowcost[i]$  表示顶点  $i$  到集合  $U$  (当前最小生成树的顶点集合) 的距离, 而  $closest[i]$  表示集合  $U$  中的某个顶点, 该顶点和顶点  $i$  组成的边的权值即为  $lowcost[i]$ 。

初始时,  $U = \{V_0\}$ , 所以  $closest[i] = V_0$  ( $i = 1, 2, \dots, n; i < > V_0$ ), 而  $lowcost[i] = cost[V_0, i]$ , 然后组织循环扫描数组  $lowcost$  寻找顶点  $k$ , 使满足:  $lowcost[k] = \min\{lowcost[i] \mid i \in V - U\}$ , 则  $(k, closest[k])$  是本次找到权值最小的边。输出: 令  $lowcost[k] = 0$ , 表示顶点  $k$  并入集合  $U$ , 在程序中并没有真正设立集合  $U$ , 但  $U$  是存在的, 某个顶点  $i$  ( $1 \leq i \leq n$ ) 满足  $lowcost[i] = 0$ , 即顶点  $i$  到  $U$  的距离为 0, 那么  $i$  就在集合  $U$  里了。

每当找到一个顶点  $k$  并入集合  $U$  后, 则要判断那些  $V - U$  集合内的顶点  $j$  到集合  $U$  的距离是否改变, 如果  $cost[k, j] < lowcost[j]$ , 则令  $lowcost[j] := cost[k, j]$  和  $closest[j] := k$ , 即顶点  $j$  到集合  $U$  的距离改由边  $(k, j)$  的权值  $cost[k, j]$  代替。如此反复寻找  $k$ , 直到网  $G$  的全部顶点都并入集合  $U$ 。

[程序说明]

Const  $nv = 20$ ;

Type  $mat = \text{array}[1..nv, 1..nv]$  of integer;

Procedure  $prim$  (var  $cost$ :  $mat$ ;  $v0$ : integer);

Var

$lowcost, closest$ :  $\text{array}[1..nv]$  of integer;

$i, j, k, min$ : integer;

Begin

for  $i := 1$  to  $nv$  do

begin  $lowcost[i] := cost[v0, i]$ ;  $closest[i] := v0$ ; end;

for  $i := ①$  do

begin

$min := 32767$ ;

for  $j := 1$  to  $nv$  do

if ②

then begin

$min := lowcost[j]$ ; ③

end;

```
writeln (closest [k]: 5, k: 5, cost [closest [k], k]: 5);
④;
for j: = 1 to nv do
if ⑤ then
begin
    lowcost [j]: = cost [k, j]; closest [j]: = k
end;
end;
End;
```

4. [问题描述] 给出一个 AOV 网, 试求出它的拓扑排序。

[算法分析] 算法步骤如下:

- ①在网中选择 一个没有入度为 0 的顶点且输出之;
- ②从网中删除该顶点, 并且删去从该顶点发出的全部有向边;
- ③重复上述两步, 直至网中不存在没有入度为 0 的顶点为止。

[程序说明]

```
Const    nax = 20;
Type     link = ^node;
          node = record
            vex: 0..nax;
            next: link
          end;
adjelist = array [1..nax] of node;
Procedure toporder (var al: adjelist; n: integer);
Var
    i, j, k, top: integer; q: link;
Begin
    top: = 0;
    for i: = 1 to n do
    if al [i] . vex = 0 then begin al [i] . vex: = top; top: = i end;
    i: = 0;
    writeln ( 'Does this network have a topological order asbelow?' );
    while top < > 0 do
    begin
        j: = top; top: = al [top] . vex; write ( '——v' ), j: 2);
        ①; q: = al [j] . next;
        while q < > ② do
        begin
```

```

    k := q.vex; al[k].vex := ③;
    if ④ then begin al[k].vex := top; top := k; end;
    q := ⑤;
  end;
writeln;
end;
if i < n then writeln ( 'No, this network has a cycle. ' ) else write ( 'Yes. ' );
End;

```

## 10.2 数据结构综合测试二

(时量: 3 小时 分数: 100 分)

一、选择题 (每小题 2 分, 共 30 分)

1. 若某链表中最常用的操作是在最后一个结点之后插入一个结点和删除最后一个结点, 则采用( )存储方式最节省运算时间。

(1) 单链表 (2) 双链表 (3) 单循环链表 (4) 带头结点的双重循环链表

2. 设一个栈的输入序列为 A, B, C, D, 则借助一个栈所得到的输出序列不可能是( )

(1) A, B, C, D (2) D, C, B, A  
(3) A, C, D, B (4) D, A, B, C

3. 串是( )。

(1) 不少于一个字母的序列 (2) 任意个字母的序列  
(3) 不少于一个字符的序列 (4) 有限个字符的序列

4. 链表不具有的特点是( )。

(1) 可随机访问任一元素 (2) 插入删除不需要移动元素  
(3) 不必事先估计存储空间 (4) 所需空间与线性表长度成正比

5. 在有 N 个叶子结点的哈夫曼树中, 其结点总数为( )。

(1) 不确定 (2)  $2n$  (3)  $2n+1$  (4)  $2n-1$

6. 任何一个无向连通图的最小生成树( )。

(1) 只有一棵 (2) 有一棵或多棵  
(3) 一定有多棵 (4) 可能不存在

7. 将一棵有 100 个结点的完全二叉树从根这一层开始, 每层上从左到右依次对结点进行编号, 根结点的编号为 1, 则编号为 49 的结点的左孩子编号为( )。

(1) 98 (2) 99 (3) 50 (4) 48

8. 下列序列中, ( ) 是执行第一趟快速排序后得到的序列 (排序的关键字类型是字符串)。

(1) [da, ax, eb, de, bb] ff [ha, gc]  
(2) [cd, eb, ax, da] ff [ha, gc, bb]  
(3) [gc, ax, eb, cd, bb] ff [da, ha]  
(4) [ax, bb, cd, da] ff [eb, gc, ha]

9. 用  $n$  个键值构造一棵二叉排序树, 最低高度为( )。

- (1)  $n/2$  (2)  $n$  (3)  $\log_2 n$  (4)  $\log_2 n + 1$

10. 二分查找法要求查找表中各元素的键值必须是( )排列。

- (1) 递增或递减 (2) 递增 (3) 递减 (4) 无序

11. 数组  $A[1..5, 1..6]$  的每个元素占 5 个单元, 将其按行优先的次序存储在起始地址为 1000 的连续的内存单元中, 则元素  $A[5, 5]$  的地址为( )。

- (1) 1145 (2) 1140 (3) 1120 (4) 1125

12. 求最短路径的 DIJKSTRA 算法的时间复杂度为( )。

- (1)  $O(n)$  (2)  $O(n+e)$   
(3)  $O(n^2)$  (4)  $O(n * e)$

13. 下列排序算法中, ( ) 每一趟都能选出一个元素放在其最终位置上, 并且是不稳定的。

- (1) 冒泡排序 (2) 希尔排序  
(3) 直接选择排序 (4) 直接插入排序

14. 队列操作的原则是( )。

- (1) 先进先出 (2) 后进先出  
(3) 只能进行插入 (4) 只能进行删除

15. 有 64 个结点的完全二叉树的深度为( ) (根的层次为 1)。

- (1) 8 (2) 7 (3) 6 (4) 5

二、判断题 (10 分, 每小题 1 分, 正确的打“√”, 错误的打“×”)

- 串长度是指串中不同字符的个数。
- 数组可以看成是线性结构的一种推广, 因此可以对它进行插入、删除等运算。
- 在顺序表中取出第  $I$  个元素所花的时间与  $I$  成正比。
- 在栈满情况下不能作进栈操作, 否则产生“上溢”。
- 二路归并排序的核心操作是将两个有序序列归并为一个有序序列。
- 对任意一个图, 从它的某个顶点出发进行一次深度优先或广度优先搜索遍历可访问到该图的每一个顶点。
- 一个有向图的邻接表和逆邻接表中的结点个数一定相等。
- 二叉排序树或者是一棵空树, 或者是具有下列性质的二叉树: 若它的左子树非空, 则根结点的值大于其左孩子的值; 若它的右子树非空, 则根结点的值小于其右孩子的值。
- 若一棵二叉树的任一非叶子结点的度为 2, 则该二叉树为满二叉树。
- 只有在初始数据表为倒序时, 冒泡排序所执行的比较次数最多。

三、填空题 (20 分, 每题 2 分)

- 设  $S[1..maxsize]$  为一个顺序存储的栈, 变量  $top$  指示栈顶位置, 栈为空的条件是\_\_\_\_\_, 栈为满的条件是\_\_\_\_\_。
- 有向图  $G$  用邻接矩阵  $A[1..n, 1..n]$  存储, 其第  $I$  行的非零元素之和等于顶点  $I$  的\_\_\_\_\_。
- 分别采用堆排序、快速排序、插入排序和归并排序算法对初始状态为递增序列的表按递增顺序排序, 最省时间的是\_\_\_\_\_算法, 最费时间的是\_\_\_\_\_算法。
- 一棵二叉树的前序序列和中序序列分别如下, 画出该二叉树。

前序序列: ABCDEFGHIJ

中序序列: CBEDAGHFJI

5. 对下面给出的数据序列, 构造一棵哈夫曼树, 并求出其带权路径长度。

{4, 5, 6, 7, 10, 12, 15, 18, 23}

6. 在有  $n$  个结点的无向图中, 其边数最多为\_\_\_\_\_。

7. 在有  $n$  个顶点的有向图  $G$  中最多有\_\_\_\_\_条弧。

8. 已知栈的输入序列为  $1, 2, 3, \dots, n$ , 输出序列为  $a_1, a_2, \dots, a_n$ ,  $a_2 = n$  的输出序列共有\_\_\_\_\_种输出序列。

9. 3 个结点可构成\_\_\_\_\_棵不同形态的树。

10. 一棵二叉树的前序、中序的后序序列分别如下, 其中有一部分未显示出来, 试求出空格处的内容, 并画出该二叉树。

前序\_ B\_ F\_ ICEH\_ G

中序 D\_ KFIA\_ EJC

后序\_ K\_ FBHJ\_ G\_ A

四、程序填空题 (40 分, 每空 2 分)

1. 已知有  $n$  个选手  $P_1, P_2, \dots, P_n$  参加一项比赛, 每对选手之间非胜即负, 试设计算法求出一个选手序列  $P_1', P_2', \dots, P_n'$ , 使得  $P_i'$  胜  $P_{(i+1)'}$  ( $i \leq n$ )。

[算法分析]

用顶点  $1..n$  表示选手; 对任意两个选手  $p_i$  和  $p_j$ , 若  $p_i$  胜  $p_j$ , 则在图中与之相对应的有一条从  $I$  到  $J$  的弧; 为简便起见, 用邻接矩阵来表示图  $A[1..n, 1..n]$ , 用数组  $B[1..n]$  来保存路径上的元素。构思如下:

(1) 路径上已有  $K$  个元素;

(2) 若  $K = N$ , 说明已求得一解, 输出, 然后返回;

(3) 若  $K < N$ , 依次从余下的元素中取出与  $B[K]$  邻接的顶点放置在  $B[K+1]$ , 转 (1);

(4) 若  $K < N$ , 而在余下的这些顶点中找不到一个  $B[K]$  的邻接点, 或者是虽然存在邻接点, 但这些结点均已在同等条件下放置过了, 因此需从路径上去掉  $B[K]$ , 转 (1)。

[过程]

```
procedure getcc (k: integer);
```

```
Begin
```

```
  If (k = n) then print (B)
```

```
  Else if (k < n) and (k > 0) then
```

```
    Begin
```

```
      For I: = 1 to n do
```

```
        If ( ① ) and ( ② )
```

```
          Then begin visited [I]: = true; ③;
```

```
          Getcc (k+1);
```

```
          ④
```

```
        End;
```

```
End;
```

End;

其中 visited [I] 为标志数组, 表示各结点是否已加入, 调用前, 该数组全置为 false。

2. 已知一类大整数不超过 100 位, 设计算法以实现两个大整数的乘法运算。

[算法分析]

这类题目一般用链表来存储, 这里由于有固定的位数, 采用数组来存储。

构思如下:

(1) 用数组来存储该数的一位数字, 并约定数组下标为 0..99, 下标为 0 的元素存储个位数, 下标为 1 的存储十位数, 依此类推;

(2) 由于未指明是否为无符号数, 故需考虑负数的情况, 因而需要设置数的符号, 不妨约定为: 正数用 1 表示, 负数用 -1 表示;

(3) 另外, 为提高运算速度, 最好能指明数的位数;

(4) 用下述方法来描述:

```
type arr100 = array [0..99] of 0..99;
```

```
    bignum = record
```

```
        digits: arr100
```

```
        sign: -1..1;
```

```
        len: 0..99
```

```
    end
```

[过程]: (设  $a * b = c$ )

```
procedure timesbignum (var a, b, c: bignum);
```

```
Begin
```

```
    If a.len * b.len = 0 then ①
```

```
    Else begin
```

```
        For I: = 0 to 99 do
```

```
            ②;
```

```
        for ib: = 0 to b.len - 1 do
```

```
            if b.digits [ib] < > 0 then
```

```
                begin
```

```
                    ic: = ib; r: = 0
```

```
                    for ia: = 0 to a.len - 1 do
```

```
                        begin
```

```
                            ③
```

```
                            r: = x div 10;
```

```
                            c.digits [ic]: = x mod 10;
```

```
                            ④;
```

```
                        end;
```

```
                            ⑤;
```

```
                    end;
```

```
                    c.len: = ic + ord (r < > 0);
```

⑥;

end;

end;

3. 给出一个有向图, 求出它的拓扑排序。

[算法分析]

(1) 输入有向边的序列, 建立邻接表;

(2) 查找邻接表中入度为 0 的顶点, 将入度为 0 的顶点压进栈;

(3) 当栈不空时:

a. 使用退栈操作, 取得栈顶元素 J, 输出 J;

b. 在邻接表中, 查找 J 的所有输出边, 将与 J 邻接的顶点 K 的入度减 1; 若 K 的入度成 0, 则 K 进栈, 再转 (3);

(4) 当栈空时, 若有向图的所有顶点都已输出, 则结束拓扑排序; 否则说明图存在有向回路, 不能进行拓扑排序。

下面给出它的程序:

[程序]

program Toporder;

Const maxn = 100;

Type nodeptr = ^n1type;

N1type = record

Num: integer;

Link: nodeptr;

End;

Chtype = record

Count: integer;

Head: nodeptr;

End;

Var

Ch: array [1..maxn] of chtype;

M, n, top: integer;

Procedure readdata;

Var l, j, u, v: integer;

P: nodeptr;

Begin

Read (m, n);

For l := 1 to n do

Begin ①; ② end;

Begin

Read (u, v);

New (p);

```

    Ch [ v ] . count; = ch [ v ] . count + 1;
    Pr. num; = v;
    ③ ; ④ ;
    End;
End;
Procedure topol;
Var l, j, k; integer;
Begin
    Top; = 0;
    For l; = 1 to n do
        If ch [ l ] . count; = top; top; = l end
    l; = 0;
    While ⑤ do
        Begin
            J; = top;
            Top; = ch [ top ] . count;
            Write ( J; 5)
            ⑥ ;
            T; = ch [ J ] . head;
            While ⑦ do
                Begin
                    ⑧ ;
                    Ch [ k ] . count; = ch [ k ] . count - 1;
                    If ch [ k ] . count = 0 then
                        Begin ch [ k ] . count; = top;
                            ⑨ ;
                        End;
                    ⑩ ;
                End;
            End;
        End;
    End;
    If l < n then writeln ( ' This network has a cycle. ' )
    End;
    { = = = = = main = = = = = }
Begin
    Readdata;
    Topol
End.

```



## 10.3 数据结构综合测试三

(时量: 4 小时 分数: 400 分)

题 目	表达式转换	约瑟夫问题	多项式加法	循环队列
文件名	Change. pas	Joseph. pas	Plus. pas	Queue. pas
输入文件名	Change. in	Joseph. in	Plus. in	Queue. in
输出文件名	Change. out	Joseph. out	Plus. out	Queue. out
时间限制	1s	1s	1s	1s

## 一、表达式转换

## 【问题描述】

输入一个的算术表达式, 要求将它由初始的后缀表达式转换成中缀表达式, 或者是由初始的中缀表达式转换成后缀表达式。

## 【输入】

第一行: Mode, St (用一个空格隔开, Mode 为 0 表示接下来的字符串是中缀表达式, 为 1 表示为后缀表达式, St 是一个由小写字母、圆括号 (仅当 Mode = 0 时) 以及四则运算符组成的最简字符串, St 中不含空符, 且不会出现两个括号中间没有运算符的情况)。

## 【输出】

一行: 式子的答案。

## 【样例】

输入	输出
0 (a+b) * (c+d*e) +g	ab+cde*+*g+

## 三、约瑟夫问题

## 【问题描述】

圆桌上围坐着  $2n$  个人。其中  $n$  个人是好人, 另外  $n$  个人是坏人。如果从第一个人开始数数, 数到第  $m$  个人, 则立即处死该人; 然后从被处死的人之后开始数数, 再将数到的第  $m$  个人处死……依此方法不断处死围坐在圆桌上的人。试问预先应如何安排这些好人与坏人的座位, 能使得在处死  $m$  个人之后, 圆桌上围坐的剩余的  $n$  个人全是好人。

## 【输入】

只一行:  $N, M$  ( $N \leq 30000, M \leq 30000$ )

**【输出】**

只一行: 依次输出问题的解。用大写字母 G 表示好人, 大写字母 B 表示坏人。

**【样例】**

输入	输出
5 3	GBBGGBBCBG

### 三、多项式加法

**【问题描述】**

由文件输入两个多项式的各项系数和指数, 试编程求出它们的和, 并以手写的习惯输出此多项式。

1. 多项式的每一项  $axb$  用  $axb$  的格式输出;

2. 两个多项式在文件中各占一行, 每行有  $2m$  个数, 依次为第一项的系数, 第一项的指数, 第二项的系数, 第二项的指数……

3. 系数按降幂方式排列, 其中  $0 \leq \text{指数} \leq 100$

**【输入】**

有两行: 分别为两个多项式的每一项的系数和指数, 格式如上所述。

**【输出】**

只一行, 相加后的多项式表达式。

**【样例】**

输入	输出
1 2 3 0 -1 1	$x^2 - x + 3$

### 四、循环队列

**【问题描述】**

对于一个有  $N$  个单元 ( $N \geq 2$ ) 组成的循环队列, 若从进入第一个元素开始, 每隔  $T1$  个时间单位进入下一个元素, 同时从进入第一个元素开始, 每隔  $T2$  个时间单位处理完一个元素并令其出队, 试编写一个程序, 求第几个元素进队时将发生溢出。

**【输入】**

只一行:  $N, T1, T2$  ( $N, T1, T2 \leq 10^6$ )

**【输出】**

只一行: 即答案, 如果不可能溢出则输出 “NO”。

**【样例】**

输入

输出

4 1 2

7

## 10.4 数据结构综合测试四

(时量: 4 小时 分数: 400 分)

题 目	后序遍历	子结点数	圣诞节快乐	公路修建
文件名	Back. pas	Node. pas	Noel. pas	Road. pas
输入文件名	Back. in	Node. in	Noel. in	Road. in
输出文件名	Back. out	Node. out	Noel. out	Road. out
时间限制	1s	1s	1s	1s

## 一、后序遍历

## 【问题描述】

已知一棵二叉树的前序遍历和中序遍历, 求这棵二叉树的后序遍历。

【输入】输入文件包括两行: 第一行为二叉树的前序遍历, 第二行为二叉树的中序遍历。输入数据保证输入正确且不含其他字符。

【输出】输出文件仅一行, 为所给二叉树的后序遍历。

## 【样例】

输入	输出
ABC BAC	BCA

注: 所有二叉树的结点都用大写字母 'A' ~ 'Z' 标号, 不同的结点用不同的字母标号。

## 二、子结点数

## 【问题描述】

学了排序二叉树后, Jelly 对这类二叉树产生了浓厚的兴趣, 他发现, 如果一个序列一定, 那么这个序列所对应的排序二叉树也是一定的。现在, Jelly 想知道, 这棵树中每个结点的子结点数是多少。

## 【输入】

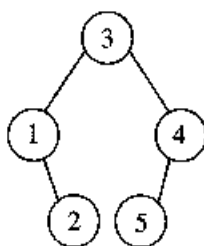
输入文件包括两行, 第一行为序列的长度  $N$ , 即序列中有多少个数, 第二行为  $N$  个 Integer 类型的数  $A_1, A_2, A_3, \dots, A_n$ , 为给定的序列。( $0 < N \leq 1000$ )

## 【输出】

输出文件包括  $N$  行, 第  $I$  行 ( $1 \leq I \leq N$ ) 有两个数, 分别为原序列中第  $I$  个数在排序二叉树中的左子树结点数和右子树结点数。

## 【样例】

输入	输出
5	2 2
3 1 2 5 4	0 1
	0 0
	1 0
	0 0



注：这里的排序方法为从小到大，即若结点的左子树非空，则它的左子树的值必小于此结点的值，对于上例对应的二叉树如右所示。

### 三、圣诞节快乐

#### 【问题描述】

圣诞节快到了，今年的圣诞节，又有成千上万的小朋友们向圣诞老人许下了自己的愿望，有很多小朋友都想得到自己喜欢的礼物，这可把圣诞老人忙坏了。由于圣诞老人有很多很多种礼物，所以他找到一件礼物需要很长时间，所以他找到礼物后，会马上把它分给想要这件礼物的所有的小朋友。为了在圣诞节那天圣诞老人能让每个小朋友的愿望得以实现，圣诞老人请你来帮他的忙。

圣诞老人给了你一个清单，上面是每个小朋友的心愿，圣诞老人已经把小朋友们按  $1 \sim N$  ( $1 \leq N \leq 65535$ ) 编了号，礼物按  $1 \sim M$  ( $1 \leq M \leq 15000$ ) 编了号，每个小朋友都有  $0 \sim X$  ( $X$  是多少，我也不知道) 件想要的礼物，这些礼物都是不同种类的。

圣诞老人需要一个清单，上面是每件礼物应该给哪些小朋友，而你必须尽快给他这个清单。

【输入】输入文件包括  $L+2$  ( $1 \leq L \leq 15000$ ) 行，第一行为  $N$  和  $M$ ，分别为小朋友的个数和礼物的种数。

第二行为  $L$ 。

第  $3 \sim (L+2)$  行每行有两个数  $A, B$ ，表示第  $A$  个小朋友想要第  $B$  种礼物。

对于第  $I+2$  行的两个数  $A_i, B_i$  和第  $J+2$  行的两个数  $A_j, B_j$  ( $1 \leq I < J \leq L$ )，输入数据保证： $(A_i < A_j) \text{ Or } ((A_i = A_j) \text{ And } (B_i < B_j))$ 。

【输出】输出文件包括  $L$  行，每行包括两个数  $B, A$ ，表示第  $B$  件礼物送给第  $A$  个小朋友。

你必须做到：

1. 对于第  $I+2$  行和两个数  $B_i, A_i$  和第  $J+2$  行的两个数  $B_j, A_j$  ( $1 \leq I < J \leq L$ ) 保证：

$(B_i < B_j) \text{ Or } ((B_i = B_j) \text{ And } (A_i < A_j))$

2. 每个小朋友都得到自己想要的礼物而不得到自己不想要的礼物。

#### 【样例】

输入	输出
4 5	1 3
8	1 4
1 2	2 1
1 3	3 1
2 3	3 2
2 4	3 3
3 1	4 2
3 3	5 4
4 1	
4 5	

## 四、公路修建

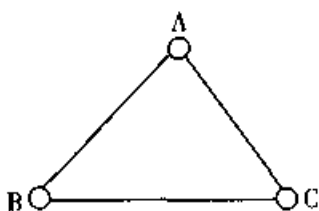
### 【问题描述】

某国有  $n$  个城市，它们互相之间没有公路相通，因此交通十分不便。为解决这一“行路难”的问题，政府决定修建公路。修建公路的任务由各城市共同完成。

修建工程分若干轮完成。在每一轮中，每个城市选择一个与它最近的城市，申请修建通往该城市的公路。政府负责审批这些申请以决定是否同意修建。

政府审批的规则如下：

- (1) 如果两个或以上城市申请修建同一条公路，则让它们共同修建；
- (2) 如果三个或以上的城市申请修建的公路成环，如下图，A 申请修建公路 AB，B 申请修建公路 BC，C 申请修建公路 CA，则政府将否决其中最短的一条公路的修建申请；



- (3) 其他情况的申请一律同意。

一轮修建结束后，可能会有若干城市可以通过公路直接或间接相连。这些可以互相连通的城市即组成“城市联盟”。在下一轮修建中，每个“城市联盟”将被看作一个城市，发挥一个城市的作用。

当所有城市被组合成一个“城市联盟”时，修建工程也就完成了。

你的任务是根据城市的分布和前面讲到的规则，计算出将要修建的公路总长度。

### 【输入】

第一行一个整数  $n$ ，表示城市的数量。（ $n \leq 5000$ ）

以下  $n$  行，每行两个整数  $x$  和  $y$ ，表示一个城市的坐标（ $-1000000 \leq x, y \leq 1000000$ ）

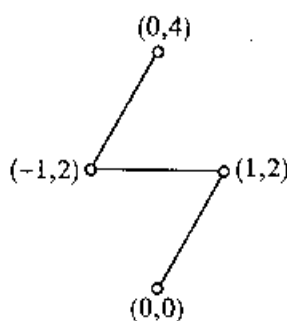
### 【输出】

一个实数，四舍五入保留两位小数，表示公路总长度。（保证有唯一解）

### 【样例】

输入	输出
4	6.47
0 0	
1 2	
-1 2	
0 4	

修建的公路如图所示：



## 10.5 数据结构综合测试一参考答案

### 一、填空题 (32 分)

- 1 (3 分) 顺序查找, 二分查找, 索引查找。
- 2 (5 分) 有穷性, 确定性, 有效性, 有 0 个或多个输入, 有 1 个或多个输出。
- 3 (3 分) 1: 1, 1: n, n: m。
- 4 (5 分) ABC, ACB, BAC, BCA, CBA。
- 5 (4 分) 先进先出, R+1, F, F
- 6 (4 分)  $- / * 3 + 5 X Y Z, 3 5 X + * Y / Z -$ 。
- 7 (4 分) 深度优先遍历:  $V_A \rightarrow V_B \rightarrow V_D \rightarrow V_F \rightarrow V_C \rightarrow V_E$ 。  
广度优先遍历:  $V_A \rightarrow V_B \rightarrow V_D \rightarrow V_F \rightarrow V_C \rightarrow V_E$ 。
- 8 (4 分) 拓扑序列:  $V_2 \rightarrow V_5 \rightarrow V_7 \rightarrow V_1 \rightarrow V_4 \rightarrow V_3 \rightarrow V_6 \rightarrow V_8$ 。

### 二、证明题 (8 分)

证明:

①树的结点总数  $n$  为:  $n = n_0 + n_1 + n_2 + \dots + n_m$

②设  $B$  为树中总的分支数目。由于树中除根结点外, 其余结点都有一个分支进入, 所以:  
 $B = n - 1$  即  $n = B + 1$

而这些分支只能由度数为 1、度数为 2、……、度数为  $m$  的结点发出, 所以:

$$B = n_1 + 2 * n_2 + 3 * n_3 + \dots + m * n_m$$

于是得:  $n = n_1 + 2 * n_2 + 3 * n_3 + \dots + m * n_m + 1$

③结合①②得到:

$$n_0 + n_1 + n_2 + \dots + n_m = n_1 + 2 * n_2 + 3 * n_3 + \dots + m * n_m + 1$$

$$\therefore n_0 = 1 + \sum_{i=1}^m (i-1) * n_i$$

### 三、程序填空题 (60 分, 每空 3 分)

1. ①  $i := a[0] . j$

②  $j := a[0] . i$

③  $t < > 0$

④  $col := a[k] . j$

⑤  $put[col] := put[col] + 1$

2. ①  $i < n - 1$   
 ②  $(r[j].data < m1) \text{ and } (r[j].tag = 0)$   
 ③  $(r[j].data < m2) \text{ and } (r[j].tag = 0)$   
 ④  $r[x1].tag = 1$   
 ⑤  $r[x2].tag = 1$
3. ① 1 to  $nv - 1$   
 ②  $(lowcost[j] < min) \text{ and } (lowcost[j] < > 0)$   
 ③  $k = j$   
 ④  $lowcost[k] = 0$   
 ⑤  $cost[k, j] < lowcost[j]$
4. ①  $i = i + 1$   
 ② nil  
 ③  $al[k].vex = 1$   
 ④  $al[k].vex = 0$   
 ⑤  $q.next$

## 10.6 数据结构综合测试二参考答案

### 一、选择题 (30 分, 每题 2 分)

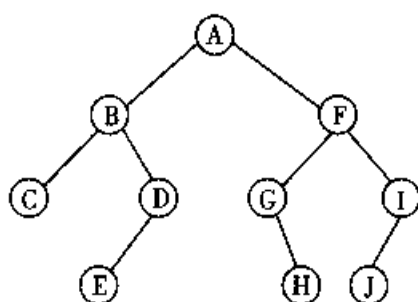
题号	1	2	3	4	5	6	7	8
答案	4	4	4	1	4	2	1	1
题号	9	10	11	12	13	14	15	
答案	4	1	1	3	3	1	2	

### 二、判断题 (10 分, 每题 1 分)

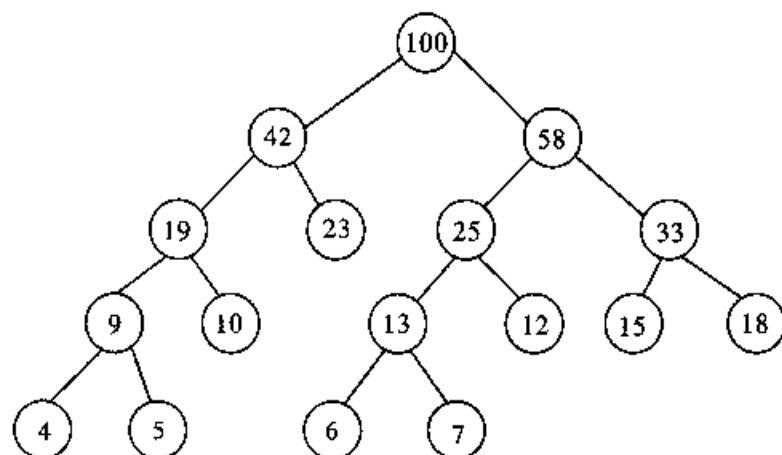
1	2	3	4	5	6	7	8	9	10
×	×	×	√	√	×	√	×	×	×

### 三、填空题 (20 分, 每题 2 分)

- $top = 0$      $top = maxsize$
- 出度
- 插入排序 快速排序
- 这棵二叉树如下:



5. WPL = 299, 哈夫曼树如下:



6.  $n(n-1)/2$

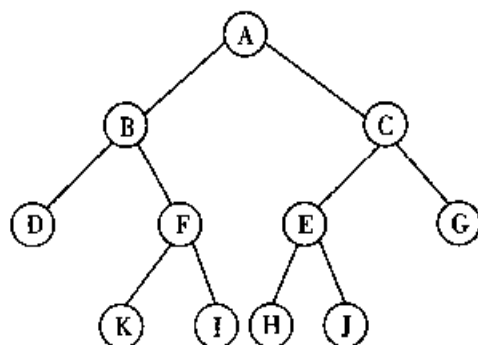
7.  $n(n-1)$

8.  $n-1$

9. 2

10. 前: ABDFKICEHJG 中: DBKFIAHEJCG 后: DKIFBHJEGCA

这棵二叉树如下:



四、程序填空题 (40 分, 每空 2 分)



1. ① not (visited [i])
- ② A [B [K], i] = I
- ③ B [k+1]: = 1
- ④ visited [i]: = false
2. ① c.len: = 0
- ② c.digits [i]: = 0
- ③ x: = c.digits [ic] + a.digits [ia] \* b.digits [ib] + r
- ④ inc (ic) 或 ic: = ic + 1
- ⑤ c.digits [ic]: = r
- ⑥ c.sign = a.sign \* b.sign
3. ① ch [i].count: = 0
- ② ch [i].head: = nil
- ③ p^.link: = ch [u].head
- ④ ch [u].head: = p
- ⑤ top < > 0
- ⑥ inc (l) 或 i: = i + 1
- ⑦ t < > nil
- ⑧ k: = t.num
- ⑨ top: = k
- ⑩ t: = t.link

## 10.7 数据结构综合测试三参考答案

### 一、表达式转换

分析:

首先介绍表达式树的概念: 对于每一个四则运算表达式, 都可以转化成一棵表达式树。转化的方法是:

1. 如果表达式中只有一个数 (没有算符), 则根结点就是这个数;
2. 如果表达式中有算符, 则找出最后运算的算符作为根结点, 把这个算符左边的部分建成左子树, 右边的部分建成右子树。

建立表达式树有什么好处呢? 可以发现, 对表达式树中序遍历就能得到中缀表达式, 对表达式树后序遍历就能得到后缀表达式。因此本题中我们只要建立表达式树, 然后进行后续遍历即可。

源程序:

```
program change;
const inputfilename = 'change.in';
      outputfilename = 'change.out';
var mode: longint;
      expression: string;
```

```

procedure solve0 (expre: string);
var len, i, p, q, bracket; longint;
    tmp: string;
    mark: boolean;
begin
    len := length (expre);
    if len = 1 then begin write (expre); exit; end;
    bracket := 0; p := 0; q := 0; mark := true;
    for i := 1 to len do
        begin
            if expre [i] = '(' then inc (bracket);
            if expre [i] = ')' then dec (bracket);
            if (bracket = 0) and (i < len) then mark := false;
            if (bracket = 0) and (expre [i] in ['+', '-']) then p := i;
            if (bracket = 0) and (expre [i] in ['*', '/']) then q := i;
        end;
    if mark then begin tmp := copy (expre, 2, len - 2); solve0 (tmp); exit; end;
    if p > 0 then i := p else i := q;
    tmp := copy (expre, 1, i - 1);
    solve0 (tmp);
    tmp := copy (expre, i + 1, len - i);
    solve0 (tmp);
    write (expre [i]);
end;

procedure solve1 (expre: string);
var len, i, count; longint;
    tmp: string;
    mark: boolean;
begin
    len := length (expre);
    if len = 1 then begin write (expre); exit; end;
    count := 0;
    for i := len - 1 downto 1 do
        begin
            if expre [i] in ['a' .. 'z'] then inc (count) else dec (count);
            if count = 1 then break;
        end;
    tmp := copy (expre, 1, i - 1);
    if (expre [len] in ['*', '/']) and (expre [i - 1] in ['+', '-']) then mark :=

```

true

```

    else mark: = false;
    if mark then write ( ' ( ' );
    solve1 ( tmp );
    if mark then write ( ' ) ' );
    //
    write ( expre [ len ] );
    //
    tmp: = copy ( expre, i, len - i );
    if not ( expre [ len - 1 ] in [ ' a ' .. ' z ' ] ) and
    ( ( expre [ len ] in [ ' * ', ' / ' ] ) or ( expre [ len - 1 ] in [ ' + ', ' - ' ] ) ) then mark:

```

= true

```

    else mark: = false;
    if mark then write ( ' ( ' );
    solve1 ( tmp );
    if mark then write ( ' ) ' );
end;
begin
    assign ( input, inputfilename ); reset ( input );
    read ( mode );
    readln ( expression );
    close ( input );
    while pos ( ' ', expression ) > 0 do
        delete ( expression, pos ( ' ', expression ), 1 );
    assign ( output, outputfilename );
    rewrite ( output );
    if mode = 0 then solve0 ( expression );
    if mode = 1 then solve1 ( expression );
    close ( output );
end.

```

## 二、约瑟夫问题

分析:

算法很简单, 按顺序数  $m$  个人, 把最后一个人标记为坏人再删掉, 重复  $n$  次即可。

最普通的解法的就是线性表“查找”法, 有两种实现方法:

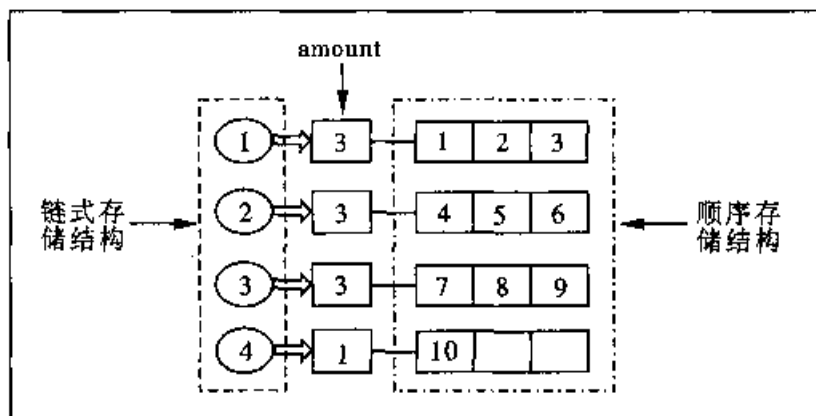
用顺序存储结构实现。用数组记录当前所有未被处死的人原来的位置, 初始值为  $1..2n$ 。

可根据前一个被处死的人在数组中的位置 (即下标) 直接定位, 找到下一个应该被处死的人在数组中的位置, 然后删去, 并将它后面的元素全部前移一次。

用链式存储结构实现。用链表记录当前所有未被处死的人原来的位置, 初始值为  $1..2n$ 。每处

死一个人后，只要将这个结点直接从链表中删去即可，然后指针后移  $(m-1)$  次，找到下一个应该被处死的人。

以上两种方法都十分简明，但缺点是效率太低。于是可以改进成“优化直接定位”法。总体思想就是在较好地实现“直接定位”的基础上，尽量避免大规模的元素移动。



设计出的数据结构如上图所示：其中 group 表示将原来的数据分为几段存储；每一段的开头记下的 amount 值表示此段中现有元素的个数。随程序的运行，amount 值是不断减小的。

这种结构可以看作是链式存储结构和顺序存储结构的结合产物，兼具这两种存储结构的优点。运用了这种存储结构后，程序效率显著提高。

源程序：

```
program joseph;
const inputfilename = 'joseph. in';
      outputfilename = 'joseph. out';
      maxn = 60000;
      maxk = 250;
type tgroup = record
      amount, st, ed: longint;
      end;
var n, m, k: longint;
    data: array [0..maxk] of tgroup;
    answer: array [1..maxn] of boolean;
procedure readdata;
begin
    assign (input, inputfilename); reset (input);
    readln (n, m);
    n := n shl 1;
    close (input);
end;
function next (q: longint): longint;
```

```

begin
    if q = k then next: = 1 else next: = q + 1;
end;
procedure main;
var i, j, p, q: longint;
begin
    k: = trunc (sqrt (n)) + 1;
    if (k - 1) * (k - 1) ≥ n then k: = k - 1;
    data [1] . st: = 1; data [1] . ed: = k;
    data [1] . amount: = k;
    for i: = 2 to k do
        begin
            data [i] . st: = data [i - 1] . ed + 1;
            data [i] . ed: = data [i] . st + k - 1;
            if data [i] . ed > n then data [i] . ed: = n;
            data [i] . amount: = data [i] . ed - data [i] . st + 1;
        end;
    p: = 0; q: = 0;
    for i: = 1 to n shr 1 do
        begin
            j: = m;
            while j > 0 do
                begin
                    if (q = data [p] . ed) and (j > data [next (p)] . amount) then
                        begin
                            p: = next (p);
                            j: = j - data [p] . amount;
                            q: = data [p] . ed;
                            if q = n + 1 then q: = 1;
                        end
                    else
                        begin
                            q: = q + 1;
                            if q > data [p] . ed then p: = p + 1;
                            if q = n + 1 then q: = 1;
                            if p = k + 1 then p: = 1;
                            if not answer [q] then j: = j - 1;
                        end;
                end;
            end;
        end;
    end;
end;

```

```

    answer [q]; = true;
end;
end;
procedure printans;
var i: longint;
begin
    assign (output, outputfilename);
    rewrite (output);
    for i: = 1 to n do
        if answer [i] then write ( 'B' ) else write ( 'G' );
    writeln;
    close (output);
end;
begin
    readdata;
    main;
    printans;
end.

```

### 三、多项式加法

分析:

多项式加法的法则是将指数相同的项合并, 合并后指数不变, 系数为原来两项的系数之和。如果某个多项式没有某一指数的项, 则可认为其系数为 0。本题中系数范围在 0 ~ 100 之内, 因此可以设立一个 0 ~ 100 的数组 count, count [i] 表示指数为 i 的项的和。初始时 count [i] = 0, 此后每读到一组  $ax^b$  中的 a, b, 就  $count[b] \leftarrow count[b] + a$ 。最后将结果输出即可。

但是输出各项时要注意一些规则: 指数为 0 的项后面不跟 x; 指数大于 0 系数为 0 的项不输出; 指数和系数都为 0 的要输出 0; 当后一项的系数为负数时, 连接两项要用减号并把后一项的系数取正。

源程序:

```

program plus;
const inputfilename = 'plus.in';
      outputfilename = 'plus.out';
      maxb = 100;
var a, b, i: longint;
    count: array [0..maxb] of longint;
    mark: boolean;
begin
    assign (input, inputfilename); reset (input);

```

```

while not seekeof do
begin
  read (a, b);
  inc (count [b], a);
end;
close (input);
assign (output, outputfilename); rewrite (output);
for i := maxb downto 0 do if count [i] < > 0 then
begin
  if mark and (count [i] > 0) then write ( ' + ');
  if count [i] < 0 then
  begin
    write ( ' - ');
    count [i] := abs (count [i]);
  end;
  mark := true;
  if (i = 0) or (count [i] > 1) then write (count [i]);
  if i > 0 then begin
    write ( ' x ');
    if i > 1 then write ( ' l ');
  end;
end;
if not mark then writeln (0);
close (output);
end.

```

#### 四、循环队列

分析:

首先我们要知道溢出有两种形式: 上溢出和下溢出。上溢出是在队列已满时插入元素, 下溢出是在队列已空时删除元素。显然:

当  $t1 = t2$  时, 队列不会溢出;

当  $t1 < t2$  时, 队列将上溢出;

当  $t1 > t2$  时, 队列将下溢出。

我们分别处理这三种情况。

当  $t1 = t2$  时不需要处理;

当  $t1 < t2$  时, 显然当某次插入元素时队列会上溢出, 即需取到最小的  $x$ , 使得:

$$x - \lfloor \frac{t1 * x}{t2} \rfloor + 1 > n$$

当  $t1 > t2$  时, 显然当某次删除元素时队列会下溢出, 继续取到最小的  $x$ , 使得:

$$x - \lfloor \frac{t2 * x}{t1} \rfloor > 1$$

直接解这两个不等式颇需要一番功夫,但幸运的是,不等式的左边随着  $x$  的增大单调变化,因此只需要二分枚举  $x$  的取值即可。

源程序:

```
! $ R -, Q -, S -, I - !
program queue;
const inputfilename = 'queue.in';
      outputfilename = 'queue.out';
var n, t1, t2: longint;
      answer: int64;
procedure solve1;
var st, ed, mid: int64;
begin
  st := 1; ed := int64 (1000000) * 1000000;
  while st <= ed do
  begin
    mid := (st + ed) shr 1;
    if mid = 7 then writeln;
    if mid - mid * t1 div t2 + 1 > n then
    begin
      ed := mid - 1;
      if answer > mid then answer := mid;
    end else st := mid + 1;
  end;
end;
procedure solve2;
var st, ed, mid: int64;
begin
  st := 1; ed := int64 (1000000) * 1000000;
  while st <= ed do
  begin
    mid := (st + ed) shr 1;
    if mid - mid * t2 div t1 > 1 then
    begin
      ed := mid - 1;
      if answer > mid then answer := mid;
    end
  end;
```





```

    else st: = mid + 1;
  end;
end;
begin
  assign (input, inputfilename); reset (input);
  readln (n, t1, t2); close (input);
  answer: = high (int64);
  if n = 0 then answer: = 0 else
    if t1 < t2 then solve1 else
      if t1 > t2 then solve2;
  assign (output, outputfilename); rewrite (output);
  if answer = high (int64) then writeln ('NO')
  else writeln (answer);
  close (output);
end.

```

## 10.8 数据结构综合测试四参考答案

### 一、后序遍历

分析:

前序遍历的特点是按照根结点、左子树、右子树的顺序遍历;中序遍历的特点是按照左子树、根结点、右子树的顺序遍历。因此如果知道前序遍历,则可知道根结点,然后找到根结点在中序遍历中的位置,左边的就是左子树,右边的就是右子树,递归处理即可。

源程序:

```

program back;
const inputfilename = 'back.in';
      outputfilename = 'back.out';
var preorder, midorder: string;
procedure solve (l, r, st: longint);
var i: longint;
begin
  if l > r then exit;
  i: = 1;
  while midorder [i] <> preorder [st] do i: = i + 1;
  solve (l, i - 1, st + 1);
  solve (i + 1, r, st + i - l + 1);
  write (preorder [st]);

```



```
end;
begin
    assign (input, inputfilename); reset (input);
    readln (preorder); readln (midorder);
    close (input);
    assign (output, outputfilename); rewrite (output);
    solve (1, length (midorder), 1);
    writeln;
    close (output);
end.
```

## 二、子结点数

分析:

本题要求模拟将  $n$  个结点插入二叉树, 然后算出每个结点左子树和右子树的结点数。由于这是一棵 BST, 并且插入顺序一定, 所以树的形态也是一定的。BST 的插入在上文已有详细的描述, 在此略去, 只考虑结点数计算的问题。每个子树的结点数可用下列递推式计算:  $\text{Sumnode} = \text{Sumlch} + \text{Sumrch}$ , 显然这个递推式需要自底向上计算, 每个结点只需要计算一次。

源程序:

```
program node;
const inputfilename = 'node.in';
      outputfilename = 'node.out';
      maxn = 1000;
var n, i, p; longint;
    key, sum, lchild, rchild; array [0..maxn] of longint;
begin
    assign (input, inputfilename); reset (input);
    readln (n);
    for i := 1 to n do
    begin
        read (key [i]);
        if i > 1 then p := 1;
        while p > 0 do
        if key [i] < key [p] then begin
            if lchild [p] = 0 then begin
                lchild [p] := i;
                break;
            end
            else p := lchild [p];
        end else
```

```

begin
  if rchild [p] = 0 then
    begin
      rchild [p] := i;
      break;
    end
  else p := rchild [p];
end;
end;
close (input);
for i := n downto 1 do
begin
  sum [i] := 1;
  if lchild [i] > 0 then inc (sum [i], sum [lchild [i]]);
  if rchild [i] > 0 then inc (sum [i], sum [rchild [i]]);
end;
assign (output, outputfilename);
rewrite (output);
for i := 1 to n do writeln (sum [lchild [i]], ' ', sum [rchild [i]]);
close (output);
end.

```

### 三、圣诞节快乐

分析:

实际上题目的输入就是输出,但是要求的顺序不同。输出要求对所有的二元组  $(A_i, B_i)$  进行 B 优先 A 次优先的排序。直接调用快速排序可以在  $O(L \log 2L)$  的时间内解决,但是输入是按照 A 优先 B 次优先给出的,因此我们用线性算法。

首先我们统计出满足  $B_i = j$  的  $i$  的二元组数目  $sum_j$ , 那么显然这  $sum_j$  个二元组的位置是连续的。可以用递推式  $st_j = 0$ ,  $st_j = ed_{j-1} + 1$ ,  $ed_j = st_j + sum_j - 1$  计算出所有  $B_i = j$  的起始和结束位置。设立若干指针  $pointer_j$ , 代表满足  $B_i = j$  的  $i$  的二元组将要插入的位置。然后开始插入二元组,  $A_i$  小的优先, 将其插到  $pointer_{B_i}$  所指的位置, 再将  $pointer_{B_i}$  加 1。

源程序:

```

program noel;
const inputfilename = 'noel.in';
      outputfilename = 'noel.out';
      maxm = 15000;
var n, m, l, i, j: longint;
    a, b, sum, st, ed, pointer, data: array [1..maxm] of longint;

```

```
begin
    assign (input, inputfilename); reset (input);
    readln (n, m); readln (l);
    for i: = 1 to l do
        begin
            readln (a [i], b [i]); inc (sum [b [i]]);
        end;
    close (input);
    st [1]: = 1; ed [1]: = sum [1];
    for i: = 2 to m do
        begin
            st [i]: = ed [i-1] + 1;
            ed [i]: = st [i] + sum [i] - 1;
        end;
    for i: = 1 to m do pointer [i]: = st [i];
    for i: = 1 to l do
        begin
            j: = b [i];
            data [pointer [j]]: = a [i];
            inc (pointer [j]);
        end;
    assign (output, outputfilename); rewrite (output);
    for i: = 1 to m do
        for j: = st [i] to ed [i] do
            writeln (i, ' ', data [j]);
        close (output);
    end.
```

#### 四、公路修建

分析:

先注意到规则 2, 可以证明这种情况绝对不会出现。设  $k$  个点  $A_1, A_2, A_3, \dots, A_k$  形成环, 也就是  $A_1$  要求建立  $A_1A_2$ ,  $A_2$  要求建立  $A_2A_3, \dots, A_k$  要求建立  $A_kA_1$ 。则必有  $A_1A_2 > A_2A_3 > \dots > A_kA_1$ , 产生矛盾, 所以这种情况不会出现。

此后可以用克鲁斯卡尔的思想证明修建方案必然是最小生成树。每次找到一条最短的边, 如果此边连接的两个点已连通, 那么根据上文不会出现环的证明, 此边必然不被选择。而如果这条边连接的两个点未连通, 由于这条边是当前最短的边, 则必然被选择。

因此我们只要求最小生成树, 由于是平面点集的最小生成树, 为了不存储所有的边, 可以考虑用普里姆算法求解。

本题关键在规则 2 看似与最小生成树矛盾, 看出其不成立, 问题便迎刃而解。

源程序:

```

program road;
const inputfilename = 'road.in';
      outputfilename = 'road.out';
      maxn = 5000;
var n: longint;
    answer: extended;
    x, y: array [1..maxn] of longint;
    shortest: array [1..maxn] of extended;
    visited: array [1..maxn] of boolean;
procedure readdata;
var i: longint;
begin
    assign (input, inputfilename); reset (input);
    readln (n);
    for i: = 1 to n do
        readln (x [i], y [i]);
    close (input);
end;
procedure main;
var i, p, j: longint;
    min, tmp: extended;
begin
    for i: = 2 to n do shortest [i]: = 1e20;
    for i: = 1 to n do
        begin
            min: = 1e22;
            for j: = 1 to n do if not visited [j] and (min > shortest [j]) then
                begin
                    p: = j; min: = shortest [j];
                end;
            visited [p]: = true;
            answer: = answer + sqrt (min);
            for j: = 1 to n do if not visited [j] then
                begin
                    tmp: = (x [p] - x [j]) * (x [p] - x [j]) + (y [p] - y [j]) * (y [p]
- y [j]);

```



```
        if shortest [j] > tmp then shortest [j] := tmp;
    end;
end;
end;
procedure printans;
begin
    assign (output, outputfilename); rewrite (output);
    writeln (answer: 0: 2);
    close (output);
end;
begin
    readdata;
    main;
    printans;
end.
```

## 习题参考答案

### 习题一：

一、选择题：CDBBDDCCB

二、1. 图形略 线性结构 2. 图形略 树状结构 3. 图形略 图结构 4. 图形略 图结构

三、略。

四、

1. 功能：判断一个数是不是素数。

复杂度：最坏情况  $O(n^{0.5})$

2. 功能：计算  $1 + 1 * 2 + 1 * 2 * 3 + 1 * 2 * 3 * 4 + \dots + 1 * 2 * \dots * n$

复杂度： $O(n)$

3. 功能：同 2

复杂度： $O(n^2)$

4. 功能：交换 A 中一些元素，使得对于任意  $i < j$ ，有  $A_i < A_j$

复杂度： $O(n^2)$

5. 功能：计算矩阵乘积

复杂度： $O(n^3)$

### 习题二：

一、单选题：BABDBAAA

二、填空题：

1.  $O(n)$ ,  $O(n)$  2. 链表，顺序 3.  $O(n)$ ,  $O(n)$  4.  $O(1)$ ,  $O(n)$

5.  $O(n)$ ,  $O(1)$  6.  $O(1)$ ,  $O(n)$  7.  $p \rightarrow next$  8.  $O(1)$ ,  $O(i)$

三、上机编程题：

1. program xt2\_3\_1;

var i, a, b, total, save: integer;

begin

total := 0; save := 0;

for i := 1 to 12 do begin

read (b);

total := total + 300 - b;

if total < 0 then begin writeln (-i); exit; end;

inc (save, total div 100);

total := total mod 100;

end;

```

        writeln (save * 120 + total);
    end;
2. program xl2_3_2;
   var name: array [1..100] of string;
       a1, a2, a5: array [1..100] of longint;
       a3, a4: array [1..100] of char;
       n, i, max, total, p: longint;
       maxname: string;
       ch: char;
       f: text;
   begin
       assign (f, 'scholar.in'); reset (f);
       readln (f, n);
       for i := 1 to n do
           begin
               read (f, ch);
               while ch < > ' ' do
                   begin
                       name [i] := name [i] + ch;
                       read (f, ch);
                   end;
               readln (f, a1 [i], a2 [i], ch, a3 [i], ch, a4 [i], ch, a5 [i]);
           end;
       close (f);
       for i := 1 to n do
           begin
               p := 0;
               if (a1 [i] > 80) and (a5 [i] >= 1) then inc (p, 8000);
               if (a1 [i] > 85) and (a2 [i] > 80) then inc (p, 4000);
               if (a1 [i] > 90) then inc (p, 2000);
               if (a1 [i] > 85) and (a4 [i] = 'Y') then inc (p, 1000);
               if (a2 [i] > 80) and (a3 [i] = 'Y') then inc (p, 850);
               if p > max then
                   begin
                       max := p;
                       maxname := name [i];
                   end;
               inc (total, p);
           end;
       end;

```



```

    assign (f, 'scholar.out'); rewrite (f);
    writeln (f, maxname); writeln (f, max);
    writeln (f, total); close (f);
end.
3. program xt2_3_3;
const
    maxlen = 350;
var
    fin, fout: text;
    necklace: array [1..maxlen*2] of char;
    len, i, j, get, max, posb, posr: integer;
function fit (a, b: char): boolean;
begin
    if (a = 'r') and (b = 'b') or (a = 'b') and (b = 'r') then fit := false else fit :=
true;
end;
begin
    assign (fin, 'heads.in'); reset (fin);
    readln (fin, len);
    for i := 1 to len do begin
        read (fin, necklace [i]);
        necklace [len + i] := necklace [i];
    end;
    close (fin);
    max := 0;
    for i := 1 to len do begin
        posb := 0; posr := 0;
        for j := 1 to len do begin
            if necklace [i + j - 1] = 'b' then if j > posb then posb := j;
            if necklace [i + j - 1] = 'r' then if j > posr then posr := j;
            if (posb > 0) and (posr > 0) then break;
        end;
        if posb = 0 then posb := len + 1;
        if posr = 0 then posr := len + 1;
        if posb > posr then get := posb - 1 else get := posr - 1;
        posb := 0; posr := 0;
        for j := 1 to len do begin
            if necklace [i + len - j] = 'b' then if j > posb then posb := j;
            if necklace [i + len - j] = 'r' then if j > posr then posr := j;

```

```

        if (posb > 0) and (posr > 0) then break;
    end;
    if posb = 0 then posb := len + 1;
    if posr = 0 then posr := len + 1;
    if posb > posr then get := get + posb - 1 else get := get + posr - 1;
    if get > len then get := len;
    if get > max then max := get;
end;
assign (fout, 'beads.out'); rewrite (fout);
writeln (fout, max);
close (fout);
end.

```

### 习题三:

一、单选题: BCDCCAABCB

二、算法设计题:

1: procedure try (i);

    Begin

        Max := max + sqr (i);

        If i < n then try (i + 1);

    End;

2: procedure try (i);

    Begin

        J := 1 mod s;

        l := 1 div s;

        If i > 0 then try (i);

        Write (J);

    End;

3: function fib (n): integer;

    Begin

        If n ≤ 2 then fib := n - 1 else fib := fib (n - 1) + fib (n - 2);

    End;

三、上机编程题:

1: program xt3\_3\_1;

uses math;

const

    inf = 'bracket.in';

    ouf = 'bracket.out';

    maxn = 200;

    none = 1 shl 30;

```

type integer = longint;
var f, g, r: array [1..maxn, 1..maxn] of integer;
    n: integer;
    st: string;

procedure main;
var i, j, k, t, tmp, cost: integer;
begin
    fillchar (f, sizeof (f), 0); fillchar (g, sizeof (g), 0);
    for i := 1 to n do begin
        f[i, i] := 2; g[i, i] := 1;
    end;
    for k := 1 to n - 1 do
        for i := 1 to n - k do begin
            j := i + k;
            f[i, j] := none;
            g[i, j] := none;
            for t := i to j - 1 do begin
                tmp := max (g[i, t], g[t + 1, j]);
                cost := f[i, t] + f[t + 1, j];
                if (cost < f[i, j]) or (cost = f[i, j]) and (tmp < g[i, j]) then begin
                    f[i, j] := cost; g[i, j] := tmp; r[i, j] := t;
                end;
            end;
        end;
    end;
    if (st[i] = '(') and (st[j] = ')') or (st[i] = '[') and (st[j] = ']') then
        if i = j - 1 then begin
            r[i, j] := 0; f[i, j] := 2; g[i, j] := 1;
        end else begin
            tmp := g[i + 1, j - 1];
            cost := f[i + 1, j - 1] + 2;
            if (cost < f[i, j]) or (cost = f[i, j]) and (tmp < g[i, j]) then begin
                f[i, j] := f[i + 1, j - 1] + 2;
                g[i, j] := g[i + 1, j - 1] + 1;
                r[i, j] := 0;
            end;
        end;
    end;
end;
end;
end;

```

```

procedure print (i, j: integer);
begin
  if i = j then
    if (st [i] = ' ( ' or (st [i] = ' ) ')) then write ( ' ( ' ) else write ( ' [ ] ' )
  else if r [i, j] = 0 then begin
    write (st [i]);
    if i + 1 ≤ j - 1 then print (i + 1, j - 1);
    write (st [j]);
  end else begin
    print (i, r [i, j]);
    print (r [i, j] + 1, j);
  end;
end;

begin
  assign (input, inf); assign (output, outf);
  reset (input); rewrite (output);
  readln (st); n := length (st);
  main; print (1, n);
  close (input); close (output);
end.

2: Program xt3_3_2;
type
  link = ^btree;
  btree = record
    info: char;
    left, right: link;
  end;
var
  p, root: link;
  str: string;
  n, i: integer;
  data: text;
procedure setbtree (var p: link);
begin
  if n >= 1 then begin
    new (p);
    p^.info := str [n];
    p^.right := nil; p^.left := nil; n := n - 1;
  end;
end;

```

```

        if p^.info in [ '+', '-', '*', '/' ] then
            begin
                setbtree (p^.right);
                setbtree (p^.left);
            end;
        end;

    end;

function btree_ travel (p: link): string;
var a, b: string;
begin
    if not (p^.info in [ '+', '-', '*', '/' ]) then btree_ travel := p^.info
    else begin
        a := btree_ travel (p^.left); b := btree_ travel (p^.right);
        if (p^.info in [ '*', '/' ]) and (p^.left^.info in [ '+', '-' ]) then a := '(' + a
        + ')';
        if (p^.info in [ '*', '-' ]) and (p^.right^.info in [ '+', '-' ]) then b := '(' +
        b + ')';
        if (p^.info in [ '/' ]) and (p^.right^.info in [ '+', '-', '*', '/' ]) then b := '('
        + b + ')';
        btree_ travel := a + p^.info + b;
    end;
end;

begin
    assign (data, 't5.txt'); reset (data);
    read (data, str); close (data); n := 0;
    for i := 1 to length (str) do
        if (str [i] = '@') and (n = 0) then n := i;
        n := n - 1;
    new (p); p^.info := str [n]; p^.left := nil; p^.right := nil;
    root := p; n := n - 1;
    if root^.info in [ '+', '-', '*', '/' ] then
        begin
            setbtree (p^.right);
            setbtree (p^.left);
        end;
    writeln (btree_ travel (root));
end.

```

3. program xt3\_ 3\_ 3;

```

var a, b; array [1..10002] of longint;
    n, i, pa, pb, qb, s, t, r; longint;
procedure quicksort (p, q: longint);
    var i, j, m, t; longint;
    begin
        if p < q then
            begin
                i := p; j := q; m := a [(i+j) div 2];
                repeat
                    while a [i] < m do inc (i);
                    while m < a [j] do dec (j);
                    if i ≤ j then
                        begin
                            t := a [i]; a [i] := a [j]; a [j] := t;
                            inc (i); dec (j);
                        end;
                until i > j;
                if p < j then quicksort (p, j);
                if i < q then quicksort (i, q);
            end;
        end;
    begin
        assign (input, 'fruit.in');
        assign (output, 'fruit.ans');
        reset (input); rewrite (output);
        readln (n);
        for i := 1 to n do
            begin
                read (a [i]);
                b [i] := maxlongint div 2;
            end;
        a [n+1] := maxlongint div 2;
        a [n+2] := maxlongint div 2;
        b [n+1] := maxlongint div 2;
        b [n+2] := maxlongint div 2;
        quicksort (1, n);
        pa := 1; pb := 1; qb := 0; s := 0;
        for i := 1 to n-1 do
            begin

```

```

t := a [pa] + a [pa+1]; r := 1;
if a [pa] + b [pb] < t then
begin
  t := a [pa] + b [pb]; r := 2;
end;
if b [pb] + b [pb+1] < t then
begin
  t := b [pb] + b [pb+1]; r := 3;
end;
inc (qb); b [qb] := t; inc (s, t);
case r of
  1: inc (pa, 2);
  2: begin inc (pa); inc (pb) end;
  3: inc (pb, 2);
end;
end;
writeln (s);
close (input); close (output);
end.

```

#### 习题四:

一、选择题: DAADDBBCC

二、阅读程序写结果:

1. 输出: zzzaaabbbcccy

2. 输出: RRRRWWWW

三、程序填空:

two [i] := 1 shl i;

s >= two [b+1] (或 k > b)

inc (m [ (s mod two [i]) + two [i] ])

m [ (i mod two [j]) + two [j] ] + m [i]

k = 1

四、上机编程题:

略

#### 习题五:

一、选择题: ACCCBD

二、上机编程题:

略。

### 习题六:

一、单选题: CACCADADCCDC

二、填空题:

1.  $N-1$  2. 5, 48 3.  $(4^h-1)/3$  4. 31, 21 5. 6 6. 2, 2, 3 7. 6

8.  $a[i*2+1]$ ,  $a[i*2]$ ,  $a[i \text{ div } 2]$  9.  $2n$ ,  $n-1$ ,  $n+1$  10. 16, 31 11. 非下降序列

12. 向上, 根结点

三、运算题:

1. 先序: abcdef 中序: cbacdf 后序: cbefda 按层: abdcef

2. 后根序列: C, B, F, E, I, J, H, G, D, A

3. 5, 3, 3, 4

4. 46 (25 (12, 37 (29)), 78 (62 (, 70)))

5. (1) (38)

(2) (38, 64)

(3) (38, 64, 52)

(4) (15, 28, 52, 64)

(5) (15, 28, 52, 64, 73)

(6) (15, 28, 40, 64, 73, 52)

(7) (15, 28, 40, 64, 73, 52, 48)

(8) (15, 28, 40, 55, 73, 52, 48, 64)

(9) (15, 26, 40, 28, 73, 52, 48, 64, 55)

(10) (12, 15, 40, 28, 26, 52, 48, 64, 55, 73)

6. (1) (15, 26, 40, 38, 64, 52, 48)

(2) (26, 38, 40, 48, 64, 52)

(3) (38, 48, 40, 52, 64)

(4) (40, 48, 64, 52)

四、上机编程题:

1. 本题可用块状链表、堆、排序二叉树、线段数等实现。

以下是促销的排序二叉树实现:

```
program xt6_4_1;
const
    inf = 'pro. in';
    ouf = 'pro. out';
    maxn = 1000000;

type
    rew = record
        l, r, data: longint
    end;

var
    a: array [1..maxn] of rew;
```



```

i, j, k, n, m, t, s1, s2, head: longint;
ans: extended;

procedure init;
begin
    assign (input, inf);
    reset (input); t: =0;
    readln (n); ans: =0
end;

procedure insert (var p: longint);
begin
    if p=0 then begin p: =t; exit end;
    if a [p] . data > a [t] . data then insert (a [p] . l) else
        insert (a [p] . r)
end;

function min (p: longint): longint;
begin
    if a [p] . l < > 0 then min: = min (a [p] . l) else min: = a [p] . data
end;

function max (p: longint): longint;
begin
    if a [p] . r < > 0 then max: = max (a [p] . r) else max: = a [p] . data
end;

procedure deletemin (var p: longint);
begin
    if a [p] . l < > 0 then deletemin (a [p] . l) else p: = a [p] . r
end;

procedure deletemax (var p: longint);
begin
    if a [p] . r < > 0 then deletemax (a [p] . r) else p: = a [p] . l
end;

procedure work;
begin
    head: =0;
    for i: =1 to n do
        begin
            read (k);
            for j: =1 to k do
                begin
                    read (m); inc (t);

```

```

        a[t].data := m; a[t].l := 0;
        a[t].r := 0; insert(head)
    end;
    s1 := min(head); s2 := max(head);
    deletemin(head); deletemax(head);
    ans := ans + s2 - s1
end
end;
procedure print;
begin
    close(input);
    assign(output, outf); rewrite(output);
    writeln(ans: 0: 0);
    close(output)
end;
begin
    init; work; print
end.

```

2. 本题即树的 LCA 问题, 有 ST, Tarjan, RMQ, 记录 2i 祖先等多种算法, 但只有  $\pm 1RMQ$  和 Tarjan 算法的时间复杂度最优, 但  $\pm 1RMQ$  编程复杂度和思考难度均较高, 所以最优算法为 Tarjan 算法。

以下是 Tarjan 算法的实现:

```

program xt6_4_2;
const
    inf = 'kom. in';
    outf = 'kom. out';
    maxn = 31000;
    maxm = 1000000;

type
    rew = record
        data, n: longint
    end;

var
    f, deep, rank: array [1..maxn] of longint;
    price: array [1..maxn] of longint;
    b, over: array [1..maxn] of boolean;
    i, j, k, n, m, tb, th: longint;
    ans: extended;

```

```

q, tree: array [1..maxn] of longint;
huge, big: array [1..maxm*2] of rew;

procedure init;
begin
    assign (input, inf);
    reset (input);
    readln (n);
    fillchar (tree, sizeof (tree), 0);
    for k: = 1 to n-1 do
    begin
        readln (i, j);
        inc (tb); big [tb] . data: = j;
        big [tb] . n: = tree [i]; tree [i]: = tb;
        inc (tb); big [tb] . data: = i;
        big [tb] . n: = tree [j]; tree [j]: = tb;
    end;
    fillchar (q, sizeof (q), 0);
    readln (m);
    readln (i);
    for k: = 2 to m do
    begin
        readln (j);
        inc (th); huge [th] . data: = j;
        huge [th] . n: = q [i]; q [i]: = th;
        inc (th); huge [th] . data: = i;
        huge [th] . n: = q [j]; q [j]: = th;
        i: = j;
    end;
    close (input);
end;

procedure dfs1 (p, d: longint);
var k: longint;
begin
    deep [p]: = d; b [p]: = false; k: = tree [p];
    while k < > 0 do
    begin
        if b [big [k] . data] then dfs1 (big [k] . data, d+1);
        k: = big [k] . n;
    end
end

```

```

end;
function head (var p; longint): longint;
begin
    if f [p] < > p then head := head (f [p]) else head := p;
    f [p] := head
end;
procedure dfs2 (p; longint);
var k, h1, h2; longint;
begin
    b [p] := false; f [p] := p;
    price [p] := p; rank [p] := 1;
    k := tree [p];
    while k < > 0 do
        begin
            if b [big [k] . data] then
                begin
                    dfs2 (big [k] . data);
                    h1 := head (p);
                    h2 := head (big [k] . data);
                    if rank [h2] > rank [h1] then
                        begin
                            f [h1] := h2; inc (rank [h2]);
                            price [h2] := price [h1]
                        end else
                            begin
                                f [h2] := h1; inc (rank [h1])
                            end
                        end;
                    k := big [k] . n
                end;
            over [p] := false;
            k := q [p];
            while k < > 0 do
                begin
                    if not (over [huge [k] . data]) then
                        begin
                            ans := ans + deep [huge [k] . data] + deep [p];
                            h1 := head (huge [k] . data);
                            ans := ans - deep [price [h1]] * 2
                        end
                    end;
                    k := q [k]
                end
            end;
            over [p] := true;
            ans := ans + price [p]
        end
    end;
end;

```

```

        end;
        k := huge [k] . n
    end
end;
procedure print;
begin
    assign (output, ouf); rewrite (output);
    writeln (ans: 0: 0);
    close (output)
end;
begin
    init;
    fillchar (b, sizeof (b), true);
    dfs1 (1, 1);
    fillchar (b, sizeof (b), true);
    fillchar (over, sizeof (over), true);
    ans := 0;
    dfs2 (1);
    print
end.

```

3. 本题可用线段树实现, 也可直接用排序 + 扫描实现。

以下是后者的实现:

```

program xt6_4_3;
const
    inf = 'pro. in';
    ouf = 'pro. out';
    maxn = 100000;
var
    a: array [1..maxn] of longint;
    b, c: array [1..maxn * 2] of longint;
    i, j, n, color, s: longint;
procedure init;
begin
    assign (input, inf);
    reset (input);
    readln (n);
    for i := 1 to n do
        begin

```

```

        readln (b [i*2-1], b [i*2], a [i]);
        c [i*2-1] := i; c [i*2] := i;
        b [i*2] := -b [i*2]
    end;
    close (input)
end;

function min (a, b: longint): boolean;
begin
    if abs (a) < abs (b) then exit (true);
    if (abs (a) = abs (b)) and (a < b) then exit (true);
    min := false
end;

function max (a, b: longint): boolean;
begin
    if abs (a) > abs (b) then exit (true);
    if (abs (a) = abs (b)) and (a > b) then exit (true);
    max := false
end;

procedure change (var a, b: longint);
var c: longint;
begin
    c := a; a := b; b := c
end;

procedure sort (l, r: longint);
var i, j, k: longint;
begin
    i := l; j := r; k := b [(l+r) div 2];
    while i < j do
    begin
        while min (b [i], k) do inc (i);
        while max (b [j], k) do dec (j);
        if i ≤ j then
        begin
            change (b [i], b [j]);
            change (c [i], c [j]);
            inc (i); dec (j)
        end
    end;
    if l < j then sort (l, j);

```

```

        if i < r then sort (i, r)
    end;
    procedure printwrong;
    begin
        assign (output, outf); rewrite (output);
        writeln ( 'WA' ); close (output);
        halt
    end;
    procedure work;
    begin
        color; := -1; s; := 0;
        for i; := 1 to n * 2 do
            if b [i] > 0 then
                if color > 0 then
                    if color < > a [c [i]] then printwrong
                        else inc (s)
                    else begin color; := a [c [i]]; s; := 1 end
                else
                    begin dec (s); if s = 0 then color; := -1 end
                end
            end;
        procedure printyes;
        begin
            assign (output, outf); rewrite (output);
            writeln ( 'Accept' ); close (output);
            halt
        end;
        begin
            init;
            sort (1, 2 * n);
            work; printyes
        end.
    end.

```

4. 本题可用并查集实现。

以下是本题的实现：

```

program   xl6_4_4;
const
    inf = 'pro. in';
    outf = 'pro. out';
    maxn = 30010;

var

```

```

f, long: array [1..maxn] of longint;
i, j, k, n, m: longint;
ch: char;

procedure init;
begin
    assign (input, inf); reset (input);
    assign (output, outf); rewrite (output);
    readln (n)
end;

function head (var p: longint): longint;
begin
    if f [p] <> p then head := head (f [p]) else head := p;
    inc (long [p], long [f [p]] - 1);
    f [p] := head
end;

procedure merge (i, j: longint);
var h1, h2: longint;
begin
    h1 := head (i); h2 := head (j);
    f [h1] := h2; long [h1] := -2;
end;

procedure qustion (i, j: longint);
var h1, h2, s1, s2: longint;
begin
    s1 := i; s2 := j;
    h1 := head (i); h2 := head (j);
    if h1 <> h2 then writeln (-1) else
        writeln (abs (long [s1] - long [s2]) - 1)
end;

procedure work;
begin
    for i := 1 to maxn do
        begin f [i] := i; long [i] := 1 end;
    for k := 1 to n do
        begin
            readln (ch, i, j);
            if ch = 'M' then merge (i, j)
            else qustion (i, j)
        end
end

```



```

end;
procedure print;
begin
    close (input); close (output)
end;
begin
    init; work; print
end.

```

## 习题七:

一、单选题: BABDCDBAC

二、填空题:

1. 2    2.  $n(n-1)/2$ ,  $n(n-1)$     3.  $n^2$     4.  $e$ ,  $2e$     5. 输出, 输入    6.  $e$ ,  $2e$   
 7.  $O(n)$ ,  $O(n)$ ,  $O(e)$     8.  $O(n^2)$ ,  $O(e)$ ,  $O(e)$     9.  $O(n^2)$ ,  $O(e)$     10.  $n$ ,  $n-1$

三、运算题:

略。

四、上机编程题:

1.

```

(1) procedure count (i: integer);
    var id: integer;
    begin
        id := 0;
        for j := 1 to n do
            if GA [i, j] then inc (id)

```

end;

(2) procedure count (i: integer);

var p: link;

id: integer;

begin

id := 0;

p := GL [i];

while p &lt;&gt; nil do begin

inc (id);

p := p<sup>^</sup>.next

end

end;

(3) procedure getmax (GA);

var maxid, id: integer;

begin

maxid := 0;

```

    for i : = 1 to n do begin
        id : = 0;
        for j : = 1 to n do
            if GA [i, j] then inc (id);
            if id > maxid then maxid : = id;
        end;
    end;

2. 这道题就是一个寻找强连通分量的问题。
program ex7_4_2;
uses math;
const
    inf = 'sea. in';
    outf = 'sea. out';
    maxn = 20000;
    maxm = 200000;
type integer = longint;
    node = record
        tow, next: integer;
    end;
var n, m, tot, now, sum, top: integer;
    g, s, num, dep, low, fath: array [1..maxn] of integer;
    a: array [1..maxn] of node;
    mark: array [1..maxn] of boolean;
procedure insert (u, v: integer);
begin
    tot : = tot + 1;
    a [tot] . next : = g [u]; a [tot] . tow : = v;
    g [u] : = tot;
end;
procedure prepare;
var i, u, v: integer;
begin
    read (n, m); tot : = 0;
    for i : = 1 to m do begin
        read (u, v);
        insert (u, v);
    end;
end;

```



```

end;
procedure dfs (u: integer);
var v, tmp: integer;
begin
    now := now + 1; top := top + 1;
    dep [u] := now; low [u] := now;
    s [top] := u; tmp := g [u];
    while tmp <> 0 do begin
        v := a [tmp] . tow;
        if dep [v] = 0 then begin
            dfs (v);
            if low [v] < low [u] then low [u] := low [v];
        end else
            if mark [v] and (low [v] < low [u]) then low [u] := low [v];
        tmp := a [tmp] . next;
    end;
    if dep [u] = low [u] then begin
        sum := sum + 1;
        num [sum] := n + 1;
        repeat
            v := s [top]; top := top - 1;
            mark [v] := false; fath [v] := sum;
            num [sum] := min (num [sum], v);
        until u = v;
    end;
end;

procedure main;
var i: integer;
begin
    fillchar (mark, sizeof (mark), true);
    fillchar (dep, sizeof (dep), 0);
    fillchar (low, sizeof (low), 0);
    top := 0; now := 0; sum := 0;
    for i := 1 to n do if mark [i] then dfs (i);
    for i := 1 to n do write (num [fath [i]], ' ');
end;

```



```
begin
assign (input, inf); assign (output, ouf);
  reset (input); rewrite (output);
  prepare;
  main;
  close (input); close (output);
end.
```

3. 可以证明, 这道题实质上就是求最小生成树, 但我们也可以二分答案然后广搜来解决。

```
program xt7_4_3;
const
  inf = 'city.in';
  ouf = 'city.out';
  maxn = 500;

type integer = longint;
  node = record
    tow, cost, next: integer;
  end;

var a: array [1..maxn * 2] of node;
  g, q: array [1..maxn] of integer;
  x: array [1..maxn] of boolean;
  n, m, tot: integer;

procedure insert (u, v, c: integer);
begin
  tot := tot + 1;
  a[tot].tow := v; a[tot].next := g[u];
  a[tot].cost := c; g[u] := tot;
end;

procedure prepare;
var i, u, v, c: integer;
begin
  fillchar (g, sizeof (g), 0);
  read (n, m); tot := 0;
  for i := 1 to m do begin
    read (u, v, c);
    insert (u, v, c);
    insert (v, u, c);
  end;
```

```

    end;
end;

function check (p: integer): boolean;
var first, last, tmp, u, v: integer;
begin
    fillchar (x, sizeof (x), true);
    first := 0; last := 1;
    q [1] := 1; x [1] := false;
    while first < last do begin
        first := first + 1; u := q [first];
        tmp := g [u];
        while tmp < > 0 do begin
            v := a [tmp] . tow;
            if (a [tmp] . cost ≤ p) and x [v] then begin
                last := last + 1;
                q [last] := v; x [v] := false;
            end;
            tmp := a [tmp] . next;
        end;
    end;
    check := last = n;
end;

procedure main;
var l, r, mid: integer;
begin
    l := 1; r := 10000;
    while l < r do begin
        mid := (l + r) shr 1;
        if check (mid) then r := mid
        else l := mid + 1;
    end;
    writeln (n - 1, ' ', l);
end;

begin
    assign (input, inf); assign (output, outf);

```

```

reset (input); rewrite (output);
prepare;
main;
close (input); close (output);
end.

```

4. 这道题应该首先求出原图的补图, 如果补图不能够被黑白染色, 则问题无解; 否则考虑到补图有很多个连通子图组成, 用背包来求出人数相差的最小值。

```

Program xt7_4_4;
const
    inf = 'input.txt';
    ouf = 'output.txt';
    maxn = 401;

type integer = longint;
    node = record
        tow, next: integer;
    end;

var hash, mark, g, clo: array [1..maxn] of integer;
    a: array [1..maxn * maxn] of node;
    s: array [0..maxn, 1..2] of integer;
    f, h: array [0..maxn, 0..maxn] of integer;
    n, m, sum1, sum2, tot: integer;

procedure insert (u, v: integer);
begin
    tot := tot + 1;
    a [tot] . tow := v; a [tot] . next := g [u];
    g [u] := tot;
end;

procedure prepare;
var u, v: integer;
begin
    read (n); tot := 0;
    fillchar (g, sizeof (g), 0);
    fillchar (hash, sizeof (hash), 0);

```



```

for u : = 1 to n do begin
    read (v); hash [u] : = u;
    while v < > 0 do begin
        hash [v] : = u;
        read (v);
    end;
    for v : = 1 to n do if hash [v] < > u then begin
        insert (u, v);
        insert (v, u);
    end;
end;
end;

function search (c, u, p: integer): boolean;
var tmp, v: integer;
begin
    if p = 1 then inc (sum1) else inc (sum2);
    mark [u] : = p; clo [u] : = c;
    search : = false;
    tmp : = g [u];
    while tmp < > 0 do begin
        v : = a [tmp] . tow;
        if mark [v] = mark [u] then exit;
        if (mark [v] = 0) and (not search (c, v, 3 - p)) then exit;
        tmp : = a [tmp] . next;
    end;
    search : = true;
end;

procedure main;
var u, v, i, j, k, x0, y0, ans: integer;
begin
    fillchar (mark, sizeof (mark), 0);
    fillchar (clo, sizeof (clo), 0);
    m : = 0;
    for u : = 1 to n do if mark [u] = 0 then begin
        sum1 : = 0; sum2 : = 0;
        m : = m + 1;
    end;
end;

```



```

if not search (m, u, 1) then begin
    writeln ( 'No solution' );
    exit;
end;
s [m, 1] := sum1; s [m, 2] := sum2;
end;
fillchar (f, sizeof (f), 0);
fillchar (h, sizeof (h), 0);
f [s [1, 1], s [1, 2]] := 1;
h [s [1, 1], s [1, 2]] := 1;
for k := 2 to m do begin
    for i := n downto 0 do
        for j := n downto 0 do if f [i, j] = k - 1 then begin
            x0 := i + s [k, 1];
            y0 := j + s [k, 2];
            if (x0 ≤ n) and (y0 ≤ n) then begin
                f [x0, y0] := k;
                h [x0, y0] := 1;
            end;
            x0 := i + s [k, 2];
            y0 := j + s [k, 1];
            if (x0 ≤ n) and (y0 ≤ n) then begin
                f [x0, y0] := k;
                h [x0, y0] := 2;
            end;
        end;
    end;
end;
x0 := -1; y0 := -1;
ans := maxint;
for i := 1 to n - 1 do begin
    j := n - i;
    if (f [i, j] > 0) and (abs (i - j) < ans) then begin
        ans := abs (i - j);
        x0 := i; y0 := j;
    end;
end;
if x0 = -1 then begin
    writeln ( 'No solution' );

```





```

    exit;
end;
fillchar (hash, sizeof (hash), 0);
sum1 := x0; sum2 := y0;
while (x0 + y0 > 0) do begin
    v := h [x0, y0]; u := f [x0, y0];
    hash [u] := v;
    x0 := x0 - s [u, v]; y0 := y0 - s [u, 3 - v];
end;
write (sum1, ' ');
for i := 1 to n do if hash [clo [i]] = mark [i] then write (i, ' '); writeln;
write (sum2, ' ');
for i := 1 to n do if hash [clo [i]] <> mark [i] then write (i, ' '); writeln;
end;

begin
    assign (input, inf); assign (output, outf);
    reset (input); rewrite (output);
    while not seekeof do begin
        prepare;
        main;
    end;
    close (input); close (output);
end.

```

5. 典型的迪杰斯特拉算法。程序略

#### 习题八:

一、单选题: CABCBDBCCBCCDB

二、填空题:

2. 1.4    3. 1.3

4. 45 (26 ( (12), (30, 38)), 64 ( (56), (73)))

5. (35, 74) ( (15, 35), (58), (76, 88))

三、运算题:

3. 86 (74 (53 (38, 46), 65 (14, 27)), 40 (34 (26), 16))

4. 46 (38 (26 (16 (14), 27, (34)), 40), 86 (74 (53, (65)))

四、上机编程题:

1. 略    2. 略。

3. 算法分析:



我们可以用线段树解决此题。

首先在  $[1..n]$  上建立线段树。对于线段树的每个结点  $[l..r]$ ，我们把其中的元素以  $a$  为键值建立一个平衡树。

对于每一次的询问区域，首先我们将其划分成为  $\log_2 n$  个子区域。然后二分枚举答案  $p$ 。对于每个答案  $p$ ，我们都可以在每个子区域所拥有的平衡树中用  $\log_2 n$  的时间找到有多少个小于它的元素。

这样做的复杂度是  $O(n \log_2 n) + q * O((\log_2 n)^3)$ 。

```

Program xt8_4_3;
{ $i-, s-, q-, r- }
const
    inf = 'input.txt';
    outf = 'output.txt';
    maxn = 101000;
type integer = longint;

var a: array [1..20, 1..maxn] of integer;
    g: array [1..100, 1..3] of integer;
    buf1: array [1..1 shl 13] of integer;
    buf2: array [1..1 shl 17] of integer;
    data: array [1..maxn] of integer;
    n, m, len: integer;

procedure create (k, l, r: integer);
var i, j, mid, now: integer;
begin
    if l = r then begin
        a[k, l] := data[l]; exit;
    end;
    mid := (l + r) shr 1;
    create (k + 1, l, mid);
    create (k + 1, mid + 1, r);

    i := 1; j := mid + 1; now := 1 - 1;
    while (i ≤ mid) or (j ≤ r) do begin
        now := now + 1;
        if (j > r) or (i ≤ mid) and (a[k + 1, i] < a[k + 1, j]) then begin
            a[k, now] := a[k + 1, i];
        end;
        i := i + 1; j := j + 1;
    end;
end;
    
```



```

        i := i + 1;
    end else begin
        a[k, now] := a[k + 1, j];
        j := j + 1;
    end;
end;
end;

procedure prepare;
var i: integer;
begin
    read(n, m);
    for i := 1 to n do read(data[i]);
    create(1, 1, n);
end;

procedure search(k, l, r, ll, rr: integer);
var mid: integer;
begin
    if (l = ll) and (r = rr) then begin
        len := len + 1;
        g[len, 1] := k; g[len, 2] := l;
        g[len, 3] := r; exit;
    end;
    mid := (l + r) shr 1;
    if rr ≤ mid then search(k + 1, l, mid, ll, rr) else
    if ll > mid then search(k + 1, mid + 1, r, ll, rr) else begin
        search(k + 1, l, mid, ll, mid);
        search(k + 1, mid + 1, r, mid + 1, rr);
    end;
end;

function calc(p: integer): integer;
var i, j, k, l, r, mid, sum: integer;
begin
    sum := 0;
    for i := 1 to len do begin
        k := g[i, 1]; l := g[i, 2]; r := g[i, 3];

```



```

if a [k, l] < p then begin
    while l < r do begin
        mid := (l + r + 1) shr 1;
        if a [k, mid] < p then l := mid
        else r := mid - 1;
    end;
    sum := sum + 1 - g [i, 2] + 1;
end;
end;
calc := sum;
end;

procedure main;
var i, ll, rr, l, r, k, mid: integer;
begin
    for i := 1 to m do begin
        read (ll, rr, k);
        len := 0; search (1, 1, n, ll, rr);
        l := 1; r := n;
        while l < r do begin
            mid := (l + r + 1) shr 1;
            if calc (a [1, mid]) < k then l := mid
            else r := mid - 1;
        end;
        writeln (a [1, l]);
    end;
end;

begin
    assign (input, inf); assign (output, outf);
    settextbuf (input, buf1);
    settextbuf (output, buf2);
    reset (input); rewrite (output);
    while not seekeof do begin
        prepare;
        main;
    end;
    close (input); close (output);
end;

```



end.

#### 4. 算法分析:

对于平面中的  $n$  个点, 我们可以借助例题 8-5 的方法求出其最下方的点的个数。

那么对于每一次询问  $[x0..x1]$  与  $[y0..y1]$ , 我们只要计算以下四种情况:

- 1) 询问满足  $x_k \leq x1$  且  $y_k \leq y1$  的点的个数  $s1$ ;
- 2) 询问满足  $x_k \leq x1$  且  $y_k \leq y0 - 1$  的点的个数  $s2$ ;
- 3) 询问满足  $x_k \leq x0 - 1$  且  $y_k \leq y1$  的点的个数  $s3$ ;
- 4) 询问满足  $x_k \leq x0 - 1$  且  $y_k \leq y0 - 1$  的点的个数  $s4$ ;

那么答案也就是  $s1 - s2 - s3 + s4$ 。

用这种方法, 问题就可以在  $O((n + m) * \log_2 n)$  的时间内解决。

```

program xt8_4_4;
{ $1-, s-, q-, r- |
const
    inf = 'input.txt';
    ouf = 'output.txt';
    maxn = 100000;
    maxm = 50000;

type integer = longint;
    node = array [1..3] of integer;

var a: array [0..maxn + maxm * 4] of node;
    s: array [1..maxn + maxm * 4] of integer;
    ans: array [1..maxm] of integer;
    n, m, len, now: integer;

var tmp1: node;

procedure swap (var i, j: node);
begin
    tmp1 := i; i := j; j := tmp1;
end;

procedure sorty (l, r: integer);
var i, j, mid: integer;
begin

```



```

i := 1; j := r; mid := a [(2 * l + r) div 3, 2];
while i ≤ j do begin
    while a [i, 2] < mid do i := i + 1;
    while a [j, 2] > mid do j := j - 1;
    if i ≤ j then begin
        swap (a [i], a [j]);
        i := i + 1; j := j - 1;
    end;
end;
if i < r then sortx (i, r);
if l < j then sortx (l, j);
end;

function compare (i, j; integer): boolean;
begin
    compare := (a [i, 1] < a [j, 1]) or ((a [i, 1] = a [j, 1]) and (a [j, 3] < > 0)
and (a [i, 3] = 0));
end;

procedure sortx (l, r; integer);
var i, j; integer;
begin
    i := l; j := r; a [0] := a [(l * 2 + r) div 3];
    while i ≤ j do begin
        while compare (i, 0) do i := i + 1;
        while compare (0, j) do j := j - 1;
        if i ≤ j then begin
            swap (a [i], a [j]);
            i := i + 1; j := j - 1;
        end;
    end;
    if i < r then sortx (i, r);
    if l < j then sortx (l, j);
end;

procedure create (x, y, k; integer);
begin
    len := len + 1;

```

```

a [len, 1] := x; a [len, 2] := y; a [len, 3] := k;
end;

```

```

procedure prepare;
var i, p, x0, y0, x1, y1: integer;
begin
  read (n, m); len := 0;
  for i := 1 to n do begin
    read (x0, y0);
    create (x0, y0, 0);
  end; len := n;
  for i := 1 to m do begin
    read (x0, y0, x1, y1);
    create (x1, y1, i);
    create (x0 - 1, y1, -i);
    create (x1, y0 - 1, -i);
    create (x0 - 1, y0 - 1, i);
  end;
  sorty (1, len);
  now := 1; p := a [1, 2]; a [1, 2] := 1;
  for i := 2 to len do begin
    if a [i, 2] > p then begin
      p := a [i, 2];
      now := now + 1;
    end;
    a [i, 2] := now;
  end;
  sortx (1, len);
end;

```

```

procedure add (k: integer);
begin
  while k ≤ now do begin
    s [k] := s [k] + 1;
    k := k + k and (k xor (k - 1));
  end;
end;

```

```

function calc (k: integer): integer;
var sum: integer;
begin
    sum := 0;
    while k > 0 do begin
        sum := sum + s[k];
        k := k - k and (k xor (k - 1));
    end;
    calc := sum;
end;

procedure main;
var i, sum, u: integer;
begin
    fillchar (ans, sizeof (ans), 0);
    fillchar (s, sizeof (s), 0);
    for i := 1 to len do
        if a[i, 3] = 0 then add (a[i, 2])
        else begin
            sum := calc (a[i, 2]);
            u := abs (a[i, 3]);
            if u = a[i, 3] then inc (ans[u], sum) else dec (ans[u], sum);
        end;
    for i := 1 to m do writeln (ans[i]);
end;

begin
    assign (input, inf); assign (output, outf);
    reset (input); rewrite (output);
    prepare;
    main;
    close (input); close (output);
    assign (output, ' '); rewrite (output);
    close (output);
end.

```

#### 5. 算法分析:

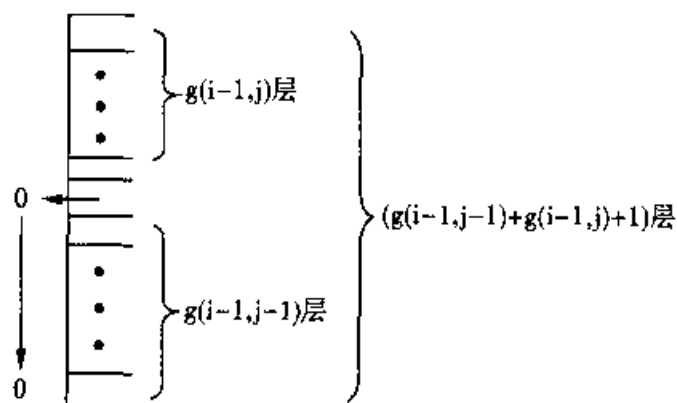
这里, 我们需要定义一个新的动态规划函数  $g(i, j)$ , 它表示用  $j$  个蛋尝试  $i$  次在最坏情况下能确定  $E$  的最高楼层数。下面具体讨论  $g(i, j)$ 。



很显然,无论有多少鹰蛋,若只试1次就只能确定一层楼,即  $g(1, j) = 1 (j \geq 1)$

而且只用1个鹰蛋试  $i$  次在最坏情况下可在  $i$  层楼中确定  $E$ , 即  $g(i, 1) = i (i \geq 1)$

状态转移也十分简单,假设第一次在某一楼层扔下一只鹰蛋,且碎了,则在后面的  $(i-1)$  次里,我们要用  $(j-1)$  个蛋在下面的楼层中确定  $E$ 。为了使  $g(i, j)$  达到最大,我们当然希望下面的楼层数达到最多,这便是一个子问题,答案为  $g(i-1, j-1)$ ; 假设第一次摔鹰蛋没碎,则在后面  $(i-1)$  次里,我们要用  $j$  个蛋在上面的楼层中确定  $E$ , 这同样需要楼层数达到最多,便为  $g(i-1, j)$  (见下图)。



因此,有如下等式成立:

$$g(i, j) = g(i-1, j-1) + g(i-1, j) + 1 \quad ⑤$$

我们的目标便是找到一个  $x$ , 使  $x$  满足  $g(x-1, M) < N$  且  $g(x, M) \geq N$  ⑥, 答案即为  $x$ ①。

这个算法乍一看是  $O(N \log_2 N)$  的, 因为  $i$  是次数, 最大为  $N$ ;  $j$  为鹰蛋数, 最大为  $M$ , 即  $\log_2 N$ , 状态转移为  $O(1)$ , 所以时间复杂度与状态数同阶, 为  $i * j$ , 即  $O(N \log_2 N)$ 。但实际情况却并非如此, 下面予以证明。

经过观察, 我们很快会发现, 函数  $g(i, j)$  与组合函数  $C$  有着惊人的相似之处, 让我们来比较一下 (见下表):

$g(i, j)$	$C_i^j$
$g(1, j) = 1 (j \geq 1)$	$C_1^1 = 1, C_1^j = 0 (j \geq 2)$
$g(i, 1) = i (i \geq 1)$	$C_i^1 = i (i \geq 1)$
$g(i, j) = g(i-1, j-1) + g(i-1, j) + 1$	$C_i^j = C_{i-1}^{j-1} + C_{i-1}^j (j \leq i), C = 0 (j > i)$

根据边界条件与递推公式, 我们可以很容易用数学归纳法证明对于任意  $i, j (i \geq 1, j \geq 1)$  总有  $g(i, j) \geq C_i^j$

又据⑥式, 可以得到

$$C_{x-1}^M \leq g(x-1, M) < N$$

① 若  $x=1$ , 须特殊判断

$$\text{即 } C_{x-1}^M < N, \frac{(x-1)(x-2)\cdots(x-M)}{M!} < N \cdots \cdots (3)$$

这里介绍一个引理：当  $1 \leq N \leq 3$  或  $N \geq 17$  时， $(\log_2 N)^2 < N$ （用函数图象可以证明）

注：当  $4 \leq n \leq 16$  时， $(\log_2 N)^2 \geq N$ ，但由于相差很小，并不影响对于渐近复杂度 2 的分析

若  $x < M$ ，则  $xM < M^2 \leq (\log_2 N)^2 < N$

若  $x \geq M$ ，则根据 (3) 式，有  $(x-M)^M \leq (x-1)(x-2)\cdots(x-M) < N \cdot M!$

$$x-M \leq M + M \sqrt[M]{NM!} \leq \sqrt[M]{NM^M} = M \sqrt[M]{N}$$

$$\therefore x \leq M + M \sqrt[M]{N} = M(1 + \sqrt[M]{N})$$

$$\therefore xM \leq M^2(1 + \sqrt[M]{N}) \cdots \cdots (4)$$

又  $\because N^{\frac{M-1}{M}} \approx N$ ，且  $N > (\log_2 N)^2 \geq M^2$ （引理）

$\therefore N^{\frac{M-1}{M}} \geq M^2$ （此性质对于  $M=1$  仍成立）

$$\log_2 N^{\frac{M-1}{M}} \geq \log_2 M^2$$

$$\frac{M-1}{M} \log_2 N \geq \log_2 M^2$$

$$\log_2 N - \frac{1}{M} \log_2 N \geq \log_2 M^2$$

$$\frac{1}{M} \log_2 N + \log_2 M^2 \leq \log_2 N$$

$$\log_2 N^{\frac{1}{M}} + \log_2 M^2 \leq \log_2 N$$

$$\text{又 } \log_2 N^{\frac{1}{M}} \approx \log_2 (N^{\frac{1}{M}} + 1)$$

$$\therefore \log_2 (N^{\frac{1}{M}} + 1) + \log_2 M^2 \leq \log_2 N$$

$$\log_2 [(N^{\frac{1}{M}} + 1) M^2] \leq \log_2 N$$

$$\therefore (N^{\frac{1}{M}} + 1) \cdot M^2 \leq N, \text{ 即 } M^2(1 + \sqrt[M]{N}) \leq N$$

又据 (4) 式，得  $xM \leq M^2(1 + \sqrt[M]{N}) \leq N$

综上所述， $xM \leq N$

这就说明了  $g(i, j)$  函数的实际运算量是  $O(N)$  级的，那么又是怎样变为  $O(\sqrt{N})$  的呢？

观察  $\frac{(x-1)(x-2)\cdots(x-M)}{M!} < N$ ，可得  $(x-1)(x-2)\cdots(x-M) < NM!$

这可以大致得出当  $M$  不太大时， $x$  与  $\sqrt[M]{N}$  是同阶的。在实际情况中，可以发现只有当  $M=1$  时， $x=N$ ， $xM=N$ ；当  $M>1$  时， $xM$  立即降至  $(\sqrt{N})$  的级别。因此，只需要在  $M=1$  时特殊判断一下就可以使算法的时间复杂度降为  $O(\sqrt{N})$  了，空间复杂度可用滚动数组降为  $O(M)$ ，即  $O(\log_2 N)$ 。

```
program xt8_4_5;
type integer = longint;
var n, m, i, j, now, last: integer;
```



```

g: array [0..1, 1..100000] of integer;

begin
  read (n, m);
  for i : = 1 to m do g [0, i] : = 1;
  now : = 1; last : = 0;
  for i : = 2 to n do begin
    g [now, 1] : = i;
    for j : = 2 to i do g [now, j] : = g [last, j - 1] + g [last, j] + 1;
    if (g [last, m] < n) and (g [now, m] >= n) then begin
      writeln (i);
      break;
    end;
    last : = now; now : = now xor 1;
  end;
end.

```

### 习题九:

一、单选题: BBACCCDBDDC

二、填空题:

1. 插入, 选择    2. 冒泡, 归并    3.  $N/2$ ,  $N-1$     4.  $\log_2 n$ ,  $n \log_2 n$   
 5. (38, 46, 56, 79, 40, 84)    6.  $n \log_2 n$ ,  $n^2$     7.  $\log_2 n$ ,  $n$     8. 略    9. 略  
 10.  $O(n)$ ,  $n \log_2 n$ ,  $n \log_2 n$     11. 5, 4, 8

三、主机编程题:

```

1. program xt9_3_1;
var
  a: array [1..1000, 1..8] of integer;
  b: array [1..1000, 1..8] of real;
  fk: array [1..1000] of real;
  c: array [1..1000] of integer;
  avg, d: array [1..8] of real;
  n: integer;

procedure init;
var i, j: integer;
begin
  assign (input, 'competition.in'); reset (input);
  assign (output, 'competition.out'); rewrite (output);
  readln (n);

```



```

fillchar (a, sizeof (a), 0);
fillchar (avg, sizeof (avg), 0);
fillchar (b, sizeof (b), 0);
fillchar (c, sizeof (c), 0);
for i: =1 to n do
begin
for j: =1 to 8 do
begin
read (a [i, j]);
avg [j]: =avg [j] + a [i, j] /n;
c [i]: =c [i] + a [i, j];
end;
readln;
end;
fillchar (d, sizeof (d), 0);
for i: =1 to n do
for j: =1 to 8 do
if a [i, j] > avg [j] then d [j]: =d [j] + (a [i, j] - avg [j]) /n
else d [j]: =d [j] + (avg [j] - a [i, j]) /n;

close (input);
end;
procedure compe;
var i, j: integer;
begin
fillchar (fk, sizeof (fk), 0);
for i: =1 to n do
for j: =1 to 8 do
if a [i, j] - avg [j] < > 0 then
begin
b [i, j]: = (a [i, j] - avg [j]) /d [j];
if j ≤ 3 then fk [i]: =fk [i] + b [i, j]
else fk [i]: =fk [i] + 0.8 * b [i, j];
end;
end;
end;
procedure outpppp;
var i, j: integer;
max, f, k: real;
pp: integer;
begin

```

```
for i: = 1 to n do
  begin
    f: = -32700;
    k: = -10000;
    for j: = 1 to n do
      if fk [j] > f then begin f: = fk [j]; k: = c [j]; pp: = j; end
      else if fk [j] = f then
        if c [j] > k then begin k: = c [j]; pp: = j; end;
    writeln (pp);
    fk [pp]: = -32310;
    c [pp]: = -11111;
  end;
close (output);
end;
begin
  init;
  compe;
  outpppp;
end.
```

2. 和例题 9-2 差不多。请参考例题去完成。

①若  $x=1$ , 须特殊判断。

②关于渐近复杂度的理论, 详情请见参考文献 [3] 第 41~49 页。